Atari BASIC Source book

http://www.abbuc.de
http://www.atari-portal.de

Converted 2006 by Andreas Bertelmann for ABBUC

Contents

Teil I	Atari BASIC: A High-Level Language Translator	3
Teil II	Internal Design	8
Teil III	Memory Usage	13
Teil IV	Program Editor	21
Teil V	Pre-Compiler	26
Teil VI	Execution	36
Teil VII	Execute Expression	40
Teil VIII	Execution Boundary Conditions	49
Teil IX	Program Flow Control Statements	52
Teil X	Tokenized Program Save and Load	56
Teil XI	The LIST and ENTER Statements	59
Teil XII	Atari Hardware Control Statements	63
Teil XIII	External Data I/O Statements	66
Teil XIV	Internal I/O Statements	71
Teil XV	Miscellaneous Statements	73
Teil XVI	Initialization	76
	Index	77

Part I

Atari BASIC: A High-Level Language Translator

Converted 2006 by Andreas Bertelmann for ABBUC

1 Atari BASIC: A High-Level Language Translator

The programming language which has become the *de facto* standard for the Atari Home Computer is the Atari 8K BASIC Cartridge, known simply as Atari BASIC. It was designed to serve the programming needs of hoth the computer novice and the experienced programmer who is interested in developing sophisticated applications programs. In order to meet such a wide range of programming needs, Atari BASIC was designed with some unique features.

In this chapter we will introduce the concepts of high level language translators and examine the design features of Atari BASIC that allow it to satisfy such a wide variety of needs.

Language Translators

Atari BASIC is what is known as a *high level language translator*.

A *language*, as we ordinarily think of it, is a system for communication. Most languages are constructed around a set of symbols and a set of rules for combining those symbols. The English language is a good example. The symbols are the words you see on this page. The rules that dictate how to combine these words are the patterns of English grammar. Without these patterns, communication would be very difficult, if not impossible: Out sentence this believe, of make don't this trying if sense you to! If we don't use the proper symbols, the results are also disastrous: @twu2 yeggopt gjsiem, keorw?

In order to use a computer, we must somehow communicate with it. The only language that our machine really understands is that strange but logical sequence of ones and zeros known as machine language. In the case of the Atari, this is known as 6502 machine language.

When the 6502 central processing unit (CPU) "sees" the sequence 01001000 in just the right place according to its rules of syntax, it knows that it should push the current contents of
the accumulator onto the CPU stack. (If you don't know what an "accumulator" or a "CPU stack" is' don't worry about it. For the discussion which follows, it is sufficient that you be aware of their existence.)

Language translators are created to make it simpler for humans to communicate with computers. There are very few 6502 programmers, even among the most expert of them, who would recognize 01001000 as the push-the-accumulator instruction. There are more 6502 programmers, but still not very many, who would recognize the hexadecimal form of 01001000, $48, as the push-the-accumulator instruction. However, most, if not all, 6502 programmers will recognize the symbol PHA as the instruction which will cause the 6502 to push the accumulator.

PHA, $48, and even 01001000, to some extent, are translations from the machine's language into a language that humans can understand more easily. We would like to be able to communicate to the computer in symbols like PHA; but if the machine is to understand us, we need a language translator to translate these symbols into machine language.

The Debug Mode of Atari's Editor/Assembler cartridge, for example, can be used to translate the symbols $48 and PHA to the ones and zeros that the machine understands. The debugger can also translate the machine's ones and zeros to $48 and PHA. The assembler part of the Editor/Assembler cartridge can be used to translate entire groups of symbols like PHA to machine code.

Assemblers

An assembler - for example, the one contained in the Assembler/Editor cartridge - is a program which is used to translate symbols that a human can easily understand into the ones and zeros that the machine can understand. In order for the assembler to know what we want it to do, we must communicate with it by using a set of symbols arranged according to a set of rules. The assembler is a translator, and the language it understands is 6502 assembly language.

The purpose of 6502 assembly language is to aid program authors in writing machine language code. The designers of the 6502 assembly language created a set of symbols and rules that matches 6502 machine language as closely as possible.

This means that the assembler retains some of the disadvantages of machine language. For instance, the process of adding two large numbers takes dozens of instructions in 6502 machine language. If human programmers had to code those dozens of instructions in the ones and zeros of machine language, there would be very few human programmers.

But the process of adding two large numbers in 6502 assembly language also takes dozens of instructions. The assembly language instructions are easier for a programmer to read and remember, but they still have a One-to-one core respondence with the dozens of machine language instructions. The programming is easier, but the process remains the same.

High Level Languages

High level languages, like Atari BASIC, Atari PILOT, and Atari Pascal, are simpler for people to use because they more closely approximate human speech and thought patterns. However, the computer still understands only machine language. So the high level languages, while seeming simple to their users, are really much more complex in their internal operations than assembly language.

Each high level language is designed to meet the specific need of some group of people. Atari Pascal is designed to implement the concept of structured programming. Atari PILOT is designed as a teaching tool. Atari BASIC is designed to serve both the needs of the novice who is just learning to program a computer and the needs of the expert programmer who is writing a sophisticated application program, but wants the program to be accessible to a large number of users.

Each of these languages uses a different set of symbols and symbol-combining rules. But all these language translators were themselves written in assembly language.

Language Translation Methods

There are two different methods of performing language translation - compilation and interpretation. Languages which translate via interpretation are called *interpreters*. Languages which translate via compilation are called *compilers*.

Interpreters examine the program source text and simulate the operations desired. Compilers translate the program source text into machine language for direct machine execution.

The compilation method tends to produce faster, more efficient programs than does the interpretation method. However, the interpretation method can make programming easier.

Problems with the Compiler Method

The compiler user first creates a program source file on a disk, using a text editing program. Then the compiler carefully examines the source program text and generates the machine language as required. Finally, the machine language code is loaded and executed. While this three-step process sounds fairly simple, it has several serious 'gotchas."

Language translators are very particular about their symbols and symbol-combining rules. If a symbol is misspelled, if the wrong symbol is used, or if the symbol is not in exactly the right place, the language translator will reject it. Since a compiler examines the enure program in one gulp, one misplaced symbol can prevent the compiler from understanding any of the rest of the program - even though the rest of the program does not violate any rules! The result is that the user often has to make several trips between the text editor and the compiler before the compiler successfully generates a machine language program.

But this does not guarantee that the program will work. If the programmer is very good or very lucky, the program will execute perfectly the very first time. Usually, however, the user must debug the program.

This nearly always involves changing the source program, usually many times. Each change in the source program sends the user back to step one: after the text editor changes the program, the compiler still has to agree that the changes are valid, and then the machine code version must be tested again. This process can be repeated dozens of times if the program is very complex.

Faster Programming or Faster Programs?
The interpretation method of language translation avoids many of these problems. Instead of translating the source code into machine language during a separate compiling step, the interpreter does all the translation *while the program is running*. This means that whenever you want to test the program you're writing, you merely have to tell the interpreter to run it. If things don't work right; stop the program, make a few changes, and run the program again at once.

You must pay a few penalties for the convenience of using the interpreter's interactive process, but you can generally develop a complex program much more quickly than the compiler user can.

However, an interpreter is similar to a compiler in that the source code fed to the interpreter must conform to the rules of the language. The difference between a compiler and an interpreter is that a compiler has to verify the symbols and symbol-combining rules only once - when the program is compiled. No evaluation goes on when the program is running. The interpreter, however, must verify the symbols and symbol-combining rules every time it attempts to run the program. If two identical programs are written, one for a compiler and one for an interpreter, the compiled program will generally execute at least ten to twenty times faster than the interpreted program.

Pre-compiling Interpreter
Atari BASIC has been incorrectly called an interpreter. It does have many of the advantages and features of an interpretive language translator, but it also has some of the useful features of a compiler. A more accurate term for Atari's BASIC Language Translator is *pre-compiIing interpreter*.

Atari BASIC, like an interpreter, has a text editor built into it. When the user enters a source line, though, the line is not stored in text form, but is translated into an intermediate code, a set of symbols called tokens. The program is stored by the editor in token form as each program line is enterred. Syntax and symbol errors are weeded out at

that time.

Then, when you run the program, these tokens are examined and their functions simulated; but hecause much of the evaluation has already been done, the execution of an Atari BASIC program is faster than-that of a pure interpreter. Yet Atari BASIC's program-building process is much simpler than that of a compiler.

Atari BASIC has advantages over compilers and interpreters alike. With Atari BASIC, every time you enter a line it is verified for language correctness. You don't have to wait until compilation; you don't even have to wait until a test run. When you type RUN you already know there are no syntax errors in your program.

Part II

Internal Design

2 Internal Design

Atari BASIC is divided into two major functional areas: the Program Constructor and the Program Executor. The Program Constructor is used when you enter and edit a BASIC program. The source line pre-compiler, also part of the Program Constructor, translates your BASIC program source text lines into tokenized lines. The Program Executor is used to execute the tokenized program - when you type RUN, the Program Executor takes over.

Both the Program Constructor and the Program Executor are designed to use data tables. Some of these tables are already contained in BASIC's ROM (read-only memory). Others are constructed by BASIC in the user RAM (random-access memory). Understanding these various tables is an important key to understanding the design of Atari BASIC.

Tokens
In Atari BASIC, tokens are the intermediate code into which the source text is translated. They represent source-language symbols that come in various lengths - some as long as 100 characters (a long variable name) and others as short as one character ("+" or "-"). Every token, however, is exactly one eight-bit byte in length.

Since most BASIC Language Symbols are more than one character long, the representation of a multicharacter BASIC Language Symbol with a single-byte token can mean a considerable saving of program storage space.

A single-byte token symbol is also easier for the Program Executor to recognize than a multi-character symbol, since it can be evaluated by machine language routines much more quickly. The SEARCH routine - 76 bytes long - located at $A462 isa good example of how much assembly language it takes to recognize a multi-character symbol. On the other hand, the two instructions located at $AB42 are enough to determine if a one-byte token is a variable. Because routines to recognize Atari BASIC'so one-byte tokens take so much less machine language, they execute relatively quickly.

The 256 possible tokens are divided into logical numerical groups that also make them simpler to deal with in assembly language. For example, any token whose value is 128 ($80) or greater represents a variable name. The logical grouping of the token values also means faster execution speeds, since, in
effect, the computer only has to check bit 7 to recognize a variable.

The numerical grouping of the tokens is shown below:

Token Value (Hex)	Description
00-0D	Unused
0E	Floating Point Numeric Constant. The next six bytes will hold its value.
0F	String Constant. The next byte is the string length. A string of that length follows.
10-3C	Operators. See table starting at $A7E3 for specific operators and values.
39-54	Functions. See table starting at $A820 for specific functions and values.
55-7F	Unused.

80-FF Variables.

In addition to the tokens listed above, there is another set of single-byte tokens, the Statement Name Tokens. Every statement in BASIC starts with a unique statement name, such as LET, PRINT, and POKE. (An assignment statement such as "A = B + C," without the word LET, is considered to begin with an implied LET.) Each of these unique statement names is represented by a unique Statement Name Token. The Program Executor does not confuse Statement Name Tokens with the other tokens because the Statement Name Tokens are always located in the same place in every statement - at the beginning. The Statement Name Token value is derived from its entry number, starting with zero, in the Statement Name Table at $A4AF.

Tables
A table is a systematic arrangement of data or infornnation. Tables in Atari BASIC fall into two distinct types: tables that are part of the Atari BASIC ROM and tables that Atari BASIC builds in the user RAM area.

ROM Tables
The following is a brief description of the various tables in the Atari BASIC ROM. The detailed use of these tables will be explained in subsequent chapters.

Statement Name Table ($A4AF). The first two bytes in each entry point to the information in the Statement Syntax Table for this statement. The rest of the entry is the name of the statement name in ATASCII. Since name lengths vary, the last character of the statement name has the most signiflcant bit turned on to indicate the end of the entry. The value of the Statement Name Token is derived from the relative (from zero) entry number of the statement name in this table.

Statement Execution Table ($AA00). Each entry in this table is the two-byte address of the 6502 machine language code which will simulate the execution of the statement. This table is organized with the statements in the same order as the statements in the Statement Name Table. Therefore, the Statement Name Token can be used as an index to this table.

Operator Name Table ($A7E3). Each entry comprises the ATASCII text of an Operator Symbol. The last character of each entry has the most significant bit turned on to indicate the end of the entry. The relative (from zero) entry number, plus 16 ($10), is the value of the token for that entry. Each of the entries is also given a label whose value is the value of the token for that symbol. For example, the ";" symbol at $A7E8 is the fifth (from zero) entry in the table. The label for the ";" token is CSC, and the value of CSC is $15, or 21 decimal (1 +5).

Operator Execution Table ($AA70). Each two-byte entry points to the address, minus one, of the routine which simulates the execution of an operator. The token value, minus 16, is used to access the entries in this table during execution time: The entries in this table are in the same order as in the Operator Name Table.

Operator Precedence Table ($AC3F). Each entry represents the relative execution precedence of an individual operator. The table entries are accessed by the operator tokens, minus 16. Entries correspond with the entries in the Operator Name Table. (See Chapter 7.)

Statement Syntax Table ($A60D). Entries in this table are used in the process of translating the source program to tokens. The address pointer in the first part of each entry in the Statement Name Table is used to access the specific syntax information for that statement in this table. (See Chapter 5.)

RAM Tables
The tables that BASIC builds in the user RAM area will be explained in detail in Chapter 3. The following is a brief description of these tables:

Variable Name Table. Each entry contains the source ATASCII text for the corresponding user variable symbol in the program. The relative (from zero) entry number of each entry in this table, plus 128, becomes the value of the token representing the variable.

Variable Value Table. Each entry either contains or points to the current value of a variable. The entries are accessed by the token value, minus 128.

Statement Table. Each entry is one tokenized BASIC program line. The tokenized lines are kept in this table in ascending numerical order by line number.

Array/String Table. This table contains the current values for all strings and numerical arrays. The location of the specific values for each string and/or array variable is accessed from information in the Variable Value Table.

Runtime Stack. This is the LIFO Runtime Stack, used to control the execution of GOSUB/RETURN and similar statements.

Pre-compiler
Atari BASIC translates the BASIC source lines from text to tokens as soon as they are entered. To do this, Atari BASIC must recognize the symbols of the BASIC Language. BASIC also requires that its symbols be combined in certain specific patterns. If the symbols don't follow the required patterns, then Atari BASIC cannot translate the line. The process of checking a source line for the required symbol patterns is called *syntax checking*.

BASIC performs syntax checking as part of the tokenizing process. When the Program Editor receives a completed line of input, the editor hands the line to the syntax routine, which examines the first word of the line for a statement name. If a valid statement name is not found, then the line is assumed to be an implied LET statement.

The grammatical rules for each statement are contained in the Statement Syntax Table. A special section of code examines the symbols in the source line, under the dfrection of the grammatical rules set forth in the Statement Syntax Table. If the source line does not conform to the rules, then it is reported back as an error. Otherwise, the line is translated to tokens. The result of this process is returned to the Program Editor for further processing.

Program Editor
When Atari BASIC is not executing statements, it is in the edit mode. When the user enters a source line and hits return, the editor accepts the line into a line buffer, where it is examined by the pre-compiler. The pre-compiler returns either tokens or an error text line.

If the line started with a line number, the editor inserts the tokenized line into the Statement Table. If the Statement Table already contains a line with the same line number, then the old line is removed from the Statement Table. The new line is then inserted just after the statement with the next lower line number and just before the statement with the next higher line number.

If the line has no line number, the editor inserts the line at the end of the Statement Table. It then passes control to the Program Executor, which will carry out the statement(s) in the line at the end of the Statement Table.

Program Executor
The Program Executor has a pointer to the statement that it is to execute. When control is passed to the executor, the pointer points to the direct (command) line at the end of the statement table. If that statement causes some other line to be executed (RUN, GOTO, GOSUB, etc.), the pointer is changed to the
new line. Lines continue to be executed as long as nothing stops that execution (END, STOP, error, etc.). When the program execution is stopped, the Program Executor returns control to the editor.

When a statement is to be executed, the Statement Name Token (the first code in the statement) directs the interpreter to the specific code that executes that statement. For instance, if that token represents the PRINT statement, the PRINT execution code is called. The execution code for each statement then examines the other tokens and simulates their operations.

Execute Expression
Arithmetic and logical expressions (A+B, C/D+E, F<G, etc.) are simulated with the Execute Expression code. Expression operators (+,-,*,etc.) have execution precedence - some operators must be executed before some others. The expression 1 + 3*4 has a value of 13 rather than 16 because * had a higher precedence than + . To properly simulate expressions, BASIC rearranges the expression with higher precedence first.

BASIC uses two temporary storage areas to hold parts of the rearranged expression. One temporary storage area, the Argument Stack, holds arguments - values consisting of constants, variables, and temporary values resulting from previous operator simulations. The other temporary storage area, the Operator Stack, holds operators. Both temporary storage areas are managed as Last-In/First-Out (LIFO) stacks.

LIFO Stacks
A LIFO (Last In/First Out) stack operates on the principle that the last object placed in the stack storage area will be the first object removed from it. If the letters A, B, C, and D, in that order, were placed in a LIFO stack, then D would be the first letter removed, followed by C, B, and A. The operations
required to rearrange the expression using these stacks will be explained in Chapter 7

BASIC also uses another LIFO stack, the Runtime Stack, in the simulation of statements such as GOSUB and FOR. GOSUB requires that BASIC remember where in the statement table the GOSUB was located so it will return to the right spot

Then RETURN is executed. If more than one GOSUB is executed before a RETURN, BASIC returns to the statement after the most recent GOSUB.

Part III

Memory Usage

3 Memory Usage

Many of BASIC's functions are controlled by a set of tables built in RAM not already occupied by BASIC or the Operating System (OS). Figure 3.1 is a diagram of memory use by both programs. Every time a BASIC programmer enters a statement, memory requirements for the RAM tables change. Memory use by the OS also varies. Different graphics modes, for example, require different amounts of memory.

These changing memory requirements are monitored, and this series of pointers keeps BASIC and the OS from overlaying each other in memory:

* High memory address (HMADR) at location $02E5
* Application high memory (APHM) at location $000E
* Low memory address (LMADR) at location $02E7

When a graphics mode requires larger screen space, the OS checks the application high memory address (APHM) that has been set by BASIC. If there is enough room for the new screen, the OS uses the upper portion of space and sets the pointer HMADR to the bottom of the screen to tell the application how much space the OS is now using.

BASIC builds its table toward high memory from low memory. The pointer to the lowest memory available to an application, called LMADR in the BASIC listing, is set by the OS to tell BASIC the lowest memory address that BASIC can use. When BASIC needs more room for one of its tables, BASIC checks HMADR. If there is enough room, BASIC uses the space and puts the highest address it has used into APHM for OS.

BASIC's operation consists primarily of building, reading, and modifying tables. Pointers to the RAM tables are kept in consecutive locations in zero page starting at $80. These tables are, in order,

* Multipurpose Buffer
* Variable Name Table
* Variable Value Table
* String/Array Table
* Statement Table
* Runtime Stack

BASIC reserves space for a buffer at LMADR. It then builds the tables contiguously (without gaps), starting at the top of the buffer and extending as far as necessary towards APHM. When a new entry needs to be added to a table, all data in the tables above is moved upward the exact amount needed to fit the new entry into the right place.

Figure 3-1. Memory usage

```
      ---------------------
FFFF  |  Operating System  |
      |        ROM         |
E000  ---------------------
      |   Floating Point   |
      |        ROM         |
D800  ---------------------
      | Hardware Registers |
D000  ---------------------
      |     Unused       |
BFFF  ---------------------
      |     BASIC ROM      |
A000  ---------------------
      |      Screen        |
```

```
             ----------------------<----HMADR

                    Free RAM

             ----------------------<---- APHM
             |        BASIC       |
             |         RAM        |
             |        Tables      |
             ----------------------<---- LMADR
             |  Operating System  |
             |         RAM        |
       0000  ----------------------
```

Variable Name Table
The Variable Name Table (VNT) is built during the pre-compile process. It is read, but not modified, during execution but only by the LIST statement. The VNT contains the names of the variables used in the program in the order in which they were entered.

The length of entries in the Variable Name Table depends on the length of the variable name. The high order bit of the last character of the name is on. For example, the ATASCII code for the variable name ABC is 414243 (expressed in hexadecimal). In the Variable Name Table it looks like this:

```
    41 42 C3
```

The $ character of a string name and the (character of an array element name are stored as part of the variable name. The table entries for variables C, AA$, and X(3) would look like this:

```
    C      C3
    AA$    41 41 A4
    X(3)   58 A8
```

It takes only two bytes to store X(3) because this table stores only X(.

A variable is represented in BASIC by a token. The value of this token is the position (relative to zero) of the variable name in the Variable Name Table, plus $80. BASIC references an entry in the table by using the token, minus $80, as an index. The Variable Name Table is not changed during execution time.

The zero page pointer to the Variable Name Table is called VNTP in the BASIC listing.

Variable Value Table
The Variable Value Table (VVT) is also built during the precompile process. It is both read and modified during execution. There is a one-to-one correspondence in the order of entries between the Variable Name Table and the Variable Value Table. If XXX is the fifth variable in the Variable Name Table, then XXX's value is the fifth entry in the Variable Value Table. BASIC references a table entry by using the variable token, minus $80, as an index

Each entry in the Variable Value Table consists of eight bytes. The first two bytes have the following meaning:

```
                   1      2
                 ---------------
                 |      |      |
```

```
                   ---------------
                     type   vnum
```

```
type = one byte, which indicates the type of variable
      $00 for floating point variable
      $40 for array variable
      $80 for string variable
vnum =      one byte, which indicates the relative position of the
variable in the tables
```

The meaning of the next six bytes varies, depending on the type of variable (floating point, string, or array). In all three cases, these bytes are initialized to zero during syntaxing and during the execution of the RUN or CLR.

When the variable is a floating point number, the six bytes represent its value.

When the variable is an array, the remaining six bytes have the following format

```
               1  2  3  4  5  6  7  8
             -------------------------
             |  |  |  |  |  |  |  |  |
             -------------------------
                   |disp | dim1| dim2
```

```
disp  =     the two-byte displacement into string/array space of
            this array variable
dim1 = two bytes indicating the first dimension value
dim2 = two bytes indicating the second dimension value
```

All three of these values are set appropriately when the array is DIMensioned during execution.

When the variable is a string, the remaining six bytes have the following meaning:

```
               1  2  3  4  5  6  7  8
             -------------------------
             |  |  |  |  |  |  |  |  |
             -------------------------
                   |disp | curl| maxl
```

disp =   the two-byte displacement into string/array space of this string variable. This value is set when the string is DIMensioned during execution.
curl =   the two-byte current length of the string. This value changes as the length of the string changes during execution.
maxl =   the two-byte maximum possible length of this string. This value is set to the DIM value during execution.

When either a string or an array is DIMensioned during execution, the low-order bit in the type byte is turned on, so that the array type is set to $41 and the string type to $81. The zero page pointer to the Variable Value Table is called VVTP in the BASIC listing.

Statement Table
The Statement Table, built as each statement is entered during editing, contains tokenized forms of the statements that were entered. This table determines what happens during execution. The format of a Statement Table entry is shown in Figure 3-2. There can be several tokens per statement and several statements per line.

Figure 3-2. Format of a Statement Table Entry

```
 ---+----------------------  ----------------------  --------------
 |     |     |     |     | .... |     |     |     | .... |     |     |
 ---+----------------------  ----------------------  --------------
 |lnum | llen| slen| snt | toks | eos |slen | snt | toks | eos | eol |
```

```
lnum  = the two-byte line number (low-order, high-order)
llen  = the one-byte line length (the displacement to the next line in
the table)
slen  = the one-byte statement length (the displacement to the next
statement in the line)
snt   = the one-byte Statement Name Token
toks  = the other tokens that make up the statement (this is variable in
length)
eos   = the one-byte end of statement token
eol   = the one-byte end of line token
```

The zero page pointer to the Statement Table is called STMTAB in the BASIC listing.

String/Array Table

The String/Array Table (also called String/Array Space) is created and modified during execution. Strings and arrays can be intermixed in the table, but they have different formats. Each array or string is pointed to by an entry in the Variable Value Table. The entry in the String/Array Table is created when the string or array is DIMensioned during execution. The data in the entry changes during execution as the value of the string or an element of the array changes.

An entry in the String/Array Table is not initialized to any particular value when it is created. The elements of arrays and the characters in a string cannot be counted upon to have any particular value. They can be zero, but they can also be garbage-data previously stored at those locations.

Array Entry

For an array, the String/Array Table contains one six-byte entry for each array element. Each element is a floating point number, stored in raveled order. For example, the entry in the String/Array Table for an array that was dimensioned as A(1,2) contains six elements, in this order: A(0,0) A(0,1) A(0,2) A(1,0) A(1,1) A(1,2)

String Entry

A string entry in the String/Array Table is created during execution, when the string is DiMensioned. The size of the entry is determined by the DIM value. The "value" of the string to BASIC at any time is determined by the data in the String/Array Table and the current length of the string as set in the Variable Value Table.

The zero page pointer to the String/Array Table is called STARP in the BASIC listing.

The Runtime Stack is created during execution. BASIC uses this LIFO stack to control processing of FOR/NEXT loops and GOSUBs. When either a FOR or a GOSUB statement is encountered during execution, an entry is put on the Runtime Stack. When a NEXT, RETURN, or a POP statement is encountered, entries are pulled off the stack.

Both the FOR entry and the GOSUB entry have a four-byte header:

```
            ----------------
```

```
             |   |   +   |   |
             -----------------
             type  lnum  disp
```

type = one byte indicating the type of element
GOSUB type = 0
FOR type = non-zero
lnum = the two-byte number of the line which contains the statement (low-order, high-order)
disp = one byte indicating the displacement into the line in the Statement Table of the token which caused this stack entry.

The FOR-type byte is actually the token representing the loop control variable from the FOR statement. (In the statement FOR I = I to 10, I is the loop control variable.) So the FOR-type byte will have a value of $80 through $FF - the possible values of a variable token. The FOR entry contains 12 additional bytes, formatted like this:

```
   1    2    3    4    5    6    7    8    9   10   11   12
 -----+----+----+----+----+---------+----+----+----+----+-----
 |                             |                             |
 -----+----+----+----+----+---------+----+----+----+----+-----
          sval                     step
```

sval = the six-byte (floating point) limit value at which to stop the loop
step = the six-byte (floating point) STEP value to increment by

The GOSUB entry consists entirely of the four-byte header. The LIST and READ statements also put a GOSUB type entry on the Runtime Stack, so that the line containing the LIST or READ can be found again when the statement has finished executing.

The zero page pointer to the Runtime Stack is called RUNSTK in the BASIC listing.

Zero Page Table Pointers
The starting addresses of the tables change dynamically during both program construction and program execution. BASIC keeps the current start addresses of the tables and other pointers required to manage memory space in contiguous zeropage cells. Each pointer is a two-byte address, low byte first.

Since these zero page cell addresses remain constant, BASIC is always able to find the tables. Here are the zero page pointers used in memory management, their names in the BASIC listing, and their addresses:

Multipurpose Buffer		$80,$81
Variable Name Table	VNTP	$82,$83
VNT dummy end	VNTD	$84,$85
Variable Value Table	VVTP	$86,$87
Statement Table	STMTAB	$88,$89
Current Statement Pointer	STMCUR	$8A,$8B
StringlArray Table	STARP	$8C,$8D
Runtime Stack	RUNSTK	$8C,$8E
Top of used memory	MEMTOP	$90,$91

Memory Management Routines
Memory Management routines allocate space to the BASIC tables as needed. There are two routines: expand, to add space, and contract, to delete space. Each routine has one entry point for cases in which the number of bytes to be added or deleted is less than 256, and another when it is greater than or equal to 256.

The EXPAND and CONTRACT routines often move many thousands of bytes each time they are called. The 6502 microprocessor is designed to move fewer than 256 bytes of data very quickly. When larger blocks of data are moved, the additional 6502 instructions required can make the process very slow. The EXPAND and CONTRACT routines circumvent this by using the less-than-256-byte fast-move capabilities in the movement of thousands of bytes. The end result is a set of very fast and very complex data movement routines.

All of this complexity does have a drawback. The infamous Atari BASIC lock-up problem lives in these two routines. If an EXPAND or CONTRACT requires that an exact multiple of 256 bytes be moved, then the routines move things from the wrong place in memory to the wrong place in memory, whereupon the computer locks up and won't respond. The only way to avoid losing hours of work this way is to SAVE to disk or cassette frequently.

EXPAND ($A881)
Parameters at entry:

```
    register
      X  =  the zero page address containing the pointer to
            the location after which space is to be added
      Y  =  the low-order part of the number of bytes to
            expand
      A  =  the high-order part of the number of bytes to
            expand
```

The routine creates a hole in the table memory, starting at a requested location and continuing the requested number of bytes. The routine first checks to see that there is enough free memory space to satisfy the request. It adds the requested expand size to each of the zero-page table pointers between the one pointed to by the X register and MEMTOP. Then each pointer will point to the correct address when EXPAND is done.

EXPAND then creates space at the address indicated by the X register. The number of bytes required is contained in the Y and A registers. (Y contains the least significant byte, while A contains the most significant.) All data from the requested address to the address pointed to by MEMTOP is moved toward high memory by the requested number of bytes. This creates a hole of the proper size.

The routine then sets Application High Memory (APHM) to the value in MEMTOP. This tells the OS the highest memory address that BASIC is currently using.

EXPLOW ($A87F)
Parameters at entry:

```
   register
      X  =  zero page address containing the pointer to the
            location after which space is to be added
      Y  =  number of bytes to expand (low-order byte only)
```

This is an additional entry point for the EXPAND routine. It is used when the number of bytes to be added to the table is less than 256. This routine first loads the 6502 accumulator with zero to indicate the most significant byte of the expand length. It then functions exactly like EXPAND.

CONTRACT ($A8FD)
Parameters at entry:

```
    register
```

X = zero page address containing the pointer to the starting location where space is to be removed
Y = the low-order part of the number of bytes to contract
A = the high-order part of the number of bytes to contract

This routine removes a requested number of bytes at a requested location by moving all the data from higher in the tables downward the exact amount needed to replace the unwanted bytes.

It subtracts the requested contract size from each of the zero page table pointers between the one pointed to by the X register and MEMTOP. Then each pointer will point to the correct address when CONTRACT is done. The routine sets application high memory (APHM) to the value in MEMTOP to indicate to the OS the highest memory address that BASIC is currently using. The block of data to be moved downward is defined by starting at the address pointed to by the zero-page address pointed to in X, plus the offset number stored in Y and A, and then continuing to the address specified at MEMTOP. Each byte of data in that block is moved downward in memory by the number of bytes specified in Y and A, effectively erasing all the data between the specified address and that address plus the requested offset.

CONTLOW ($A8FB)
Parameters at entry:
register
X = the zero page address containing the pointer to the location at which space is to be removed
Y = the number of bytes to contract (low-order byte only)

This routine is used to remove fewer than 256 bytes from the tables at a requested location by moving all the data from higher in the tables downward the exact amount needed to replace the unwanted bytes. This routine first loads the 6502 accumulator with zero to serve as the most significant byte of the contract length. It then functions exactly like CONThACT.

Miscellaneous Memory Allocations
Besides the tables, which change dynamically, BASIC also uses buffers and stacks at fixed locations. The Argument/Operator Stack is allocated at BASIC's low memory address and occupies 256 bytes. During pre-compiling it is used as the output buffer for the tokens. During execution, it is used while evaluating an expression. This buffer/stack is referenced by a pointer at location $80. This pointer has several names in the BASIC listing: LOMEM, ARGOPS, ARGSTK, and OUTBUFF.

The Syntax Stack is used during the process of syntaxing a statement. It is referenced directly that is, not through a pointer. It is located at $480 and is 256 bytes long. The Line Buffer is the storage area where the statement is placed when it is EN'TERed. It is the input buffer for the edit and pre-compile processes. It is 128 bytes long and is referenced directly as LBUFF. Often the address of LBUFF is also put into INBUFF so that the buffer can be referenced through a pointer, though INBUFF can point to other locations during various phases of BASIC's execution.

Part IV

Program Editor

4 Program Editor

The Atari keyboard is the master Control panel for Atari BASIC. Everything BASIC does has its origins at this control panel. The Program Editor's job is to service the control panel and respond to the commands that come from it. The editor gets a line from the user at the keyboard; does some preliminary processing on the line; passes the line to the pre-compiler for further processing; inserts, deletes, or replaces the line in the Statement Table; calls the Program Executor when necessary; and then waits to receive the user's next line input.

Line Processing The Program Editor, which starts at $A060, begins its process by resetting the 6502 CPU stack. Resetting the CPU stack is a drastic operation that can only occur at the beginning of a logical process. Each time Atari BASIC prepares to get a new line from the user, it restarts its entire logical process.

Getting a Line The Program Editor gets a user's line by calling CIO. The origin of the line is transparent to the Program Editor. The line may have been typed in at the keyboard or entered from some external device like the disk (if the ENTER command was given). The Program Editor simply calls CIO and asks it to put a line of not more than 255 bytes into the buffer pointed to by INBUFF (sF3). INBUFF points to the 128-byte area defined at LBUFF ($580). The OS'S screen editor, which is involved in getting a line from the keyboard, will not pass BASIC a line that is longer than 120 bytes. Normally, then, the 128-byte buffer at LBUFF is big enough to contain the user's line. Sometimes, however, if a line was originally entered from the keyboard with few blanks and many abbreviations, then LISTed to and re-ENTERed from the disk, an input line may be longer than 128 bytes. When this happens, data in the $600 page is overlaid. A LINE TOO LONG error will not necessarily 25 occur at this point. A LINE TOO LONG error occurs only if the Pre-compiler exceeds its stack while processing the line or if the tokenized line OUTBUFF exceeds 256 bytes. These overflows depend on the complexity of the line rather than on its actual length. When CIO has put a line into the line buffer (LBUFF) and the Program Editor has regained control, it checks to see if the user has changed his mind and hit the break key. If the user did indeed hit break, the Program Editor starts over and asks CIO for another line.

Flags and Indices In order to help control its processing, the Program Editor uses flags and indices. These must be given initial values.

CIX and COX. The index CIX (sF2) is used to access the user's input line in the line buffer (LBUFF), while COX ($94) is used to access the tokenized statement in the output buffer (OUTBUFF). These buffers and their indices are also used by the pre-compiler. The indices are initialized to zero to indicate the beginning of the buffers.

DIRFLG. This flag byte ($A6) is used by the editor to remember whether a line did or did not have a line number, and also to remember if the pre-compiler found an error in that line. DIRFLG is initialized to zero to indicate that the line has a line number and that the pre-compiler has not found an error.

MAXCIX. This byte ($9F) is maintained in case the line contains a syntax error. It indicates the displacement into LBUFF of the error. The character at this location will then be displayed in inverse video. The Program Editor gives this byte the same initial value as CIX, which is zero.

SVVNTP. The pointer to the current top of the Variable Name Table (VNTD) is saved as SVVNTP (SAD) so that if there is a syntax error in this line, any variables that were added can be removed. If a user entered an

erroneous line, such as 100 A=XAND B, the variable XAND would already have been added to the variable tables before the syntax error was discovered. The user probably meant to enter 100 A=X AND B, and, since there can only be 128 variables in BASIC, he probably does not want the variable XAND using up a place in the variable tables. The Program Editor uses SVVNTP to find the entry in the Variable Name Table soit can be removed. 26

SVVYTE. The process used to indicate which variable entries to remove from the Variable Value Table in case of error is different. The number of new variables in the line (SVVVTE1$B1) is initialized to zero. The Program Pre-compiler increments the value every time it adds a variable to the Variable Value Table. If a syntax error is detected, this number is multiplied by eight (the number of bytes in each entry on the Variable Value Table) to get the number of bytes to remove, counting backward from the most recent value entered.

Handling Blanks In many places in the BASIC language, blanks are not significant. For example, 100 IFX=6THENGOTO500 has the same meaning as 100 IF X = 6 THEN GOTO 500. The Program Editor, using the SKIPBLANK routine ($DBA1), skips over unnecessary blanks.

Processing the Line Number Once the editor has skipped over any leading blanks, it begins to examine the input line, starting with the line number. The floating point package is called to determine if a line number is present, and, if so, to convert the ATASCII line number to a floating point number. The floating point number is converted to an integer, saved in TSLNUM for later use, and stored in the tokenized line in the output buffer (OUTBUFF). The routine used to store data into OUTBUFF is called :SETCODE ($A2C8). When :SETCODE stores a byte into OUTBUFF, it also increments COX, that buffer's index. BASIC could convert the ATASCII line number directly to an integer, but the routine to do this would not be used any other time. Routines to convert ATASCII to floating point and floating point to integer already exist in BASIC for other purposes. Using these existing routines conserves ROM space. An interesting result of this sequence is that it is valid to enter a floating point number as a line number. For example, 100.1, 10.9, or 2.05E2 are valid line numbers. They would be converted to 100, 11, and 205 respectively If the input line does not start with a line number, the line is considered to be a direct statement. DIRFLG is set to $80 so 27

Chapter Four that the editor can remember this fact. The line number is set to 32768 ($8000). This is one larger than the largest line number a user is allowed to enter. BASIC later makes use of this fact in processing the direct statement.

Line length. The byte after the line number in the tokenized line in OUTBUFF is reserved so that the line length (actually the displacement to the next line) can be inserted later. (See Chapter 2.) The routine :SETCODE is called to reserve the byte by incrementing (COX) to indicate the next byte.

Saving erroneous lines. In the byte labeled STMSTART, the Program Editor saves the index intQ the line buffer (LBUFF) of the first non-blank character after the line number. This index is used only if there is a syntax error, so that all the characters in the erroneous line can be moved into the tokenized line buffer and from there into the Statement Table. There are advantages to saving an erroneous line in the Statement Table, because you can LIST the error line later. The advantage is greatest, not when entering a program at the keyboard, but when entering a program originally written in a different BASIC on another machine (via a modem, perhaps). Then, when a line that is not correct in Atari BASIC is entered, the line is flagged and stored - not discarded. The

user can later list the program, find the error lines, and re-enter them with the correct syntax for Atari BASIC.

Deleting lines. If the input line consists solely of a line number, the Program Editor deletes the line in the Statement Table which has that line number. The deletion is done by pointing to the line in the Statement Table, getting its length, and calling CONTRACT. (See Chapter 3.)

Statement Processing The user's input line may consist of one or more statements. The Program Editor repeats a specific set of functions for each statement in the line.

Initializing The current index (COX) into the output buffer (OUTBUFF) is saved in a byte called STMLBD. A byte is reserved in OUTBUFF by the routine :SETCODE. Later, the value in 28 STMLBD will be used to access this byte, and the statement length (the displacement to the next statement) will be stored here.

Recognizing the Statement Name After the editor calls SKBLANK to skip blanks, it processes the statement name, now pointed to by the input index (CIX). The editor calls the routine SEARCH ($A462) to look for this statement name in the Statement Name Table. SEARCH saves the table entry number of this statement name into location STENUM. The entry number is also the Statement Name Token value, and it is stored into the tokenized output buffer (OUTBUFF) as such by :SETCODE. The SEARCH routine also saves the address of the entry in SRCADR for use by the pre-compiler. If the first word in the statement was not found in the Statement Name Table, the editor assumes that the statement is an implied LET, and the appropriate token is stored. It is left to the pre-compiler to determine if the statement has the correct syntax for LET The editor now gives control to the pre-compiler, which places the appropriate tokens in OUTBUFF, increments the indices CIX and COX to show current locations, and indicates whether a syntax error was detected by setting the 6502 carry flag on if there was an error and clearing the carry flag if there was not. (See Chapter 5.)

If a Syntax Error Is Detected If the 6502 carry flag is set when the editor regains control, the editor does error processing. In MAXCIX, the pre-compiler stored the displacement into LBUFF at which it detected the error. The Program Editor changes the character at this location to inverse video. The character in inverse video may not be the point of error from your point of view, but it is where the pre-compiler detected an error. For example, assume you entered X YAND Z. You probably meant to enter X Y AND Z, and therefore would consider the error to be between Y and AND. However, since YAND is a valid variable name, X=YAND is a valid BASIC statement. The pre-compiler doesn't know there is an error until it encounters B. The value of highlighting the error with inverse 29 video is that it gives the user an approximation of where the error is. This can be a big advantage, espedally if the input line contained multiple statements or complex expressions. The next thing the editor does when a syntax error has been detected is set a value in DIRFLG to indicate this fact for future reference. Since the DIRE LG byte also indicates whether this is a direct statement, the error indicator of $40 is ORed with the value already in DIRELG. The editor takes the value that it saved in STMSTRT and puts it into CIX so that CIX now points to the start of the first statement in the input line in LBUFF. STMLBD is set to indicate the location of the first statement length byte in OUTBUFF. (A length will be stored into OUTBUFF at this displacement at a later time.) The editor sets the index into OUTBUFF (COX) to indicate the Statement Name Token of the first statement in OUTBUFF, and stores a token at that location to indicate that this line has a syntax error. The entire line (after the line number) is moved into OUTBUFF. At this point COX indicates the end of the line in OUTBUFF. (Later, the contents

of OUTBUFF will be moved to the Statement Table.) This is the end of the special processing for an erroneous line. The process that follows is done for both correct and erroneous lines.

Final Statement processing During initial line processing, the Program Editor saved in STMLBD a value that represents the location in OUTBUFF at which the statement length (displacement to the next statement) should be stored. The Program Editor now retrieves that value from STMLBD. Using this value as an index, the editor stores the value from COX in OUTBUFF as the displacement to the next statement The Program Editor checks the next character in LBUFF. If this character is not a carriage return (indicating end of the line), then the statement processing is repeated. When the carriage return is found, COX will be the displacement to the next line. The Program Editor stores COX as the line length at a displacement of two into OUTBUFF. 30

Statement Table Processing The final tokenized form of the line exists in OUTBUFF at this point. The Program Editor's next task is to insert or replace the line in the Statement Table. The Program Editor first needs to create the correct size hole in the Statement Table. The editor calls the GETSTMT routine ($A9A2) to find the address where this line should go in the Statement Table. If a line with the same line number already exists, the routine returns with the address in STMCUR and with the 6502 carry flag off. Otherwise, the routine puts the address where the new line should be inserted in the Statement Table into STMCUR and turns on the 6502 carry flag. (See Chapter 6.) If the line does not exist in the Statement Table, the editor loads zero into the 6502 accumulator. If the line does exist, the editor calls the GETLL routine ($A9DD) to put the line length into the accumulator. The editor then compares the length of the line already in the Statement Table (old line) with the length of the line in OUTBUFF (new line). If more room is needed in the Statement Table, the editor calls the EXPLOW ($A87F; see Chapter 3). If less space is needed for the new line, it calls a routine to point to the next line (GNXTL, at location $A9D0; see Chapter 6), and then calls the CONTLOW ($A8FB; see Chapter 3). Now that we have the right size hole, the tokenized line is moved from OUTBUFF into the Statement Table at the location indicated by STMCUR.

Line Wrap-up After the line has been added to the Statement Table, the editor checks DIRFLG for the syntax error indicator. If the second most significant bit ($40) is on, then there is an error.

Error Wrap-up If there is an error, the editor removes any variables that were added by this line by getting the number of bytes that were added to the Variable Name Table and the Variable Value Table from SVVNTP and SVVVTE. It then calls CONTRACT ($A8FD) to remove the bytes from each table. Next, the editor lists the line. The Statement Name Token, which was set to indicate an error, causes the word "ERROR" 31 to be printed. An inverse video character indicates where the error was detected. The editor now waits for you to enter another line.

Handling Correct Lines If the line was syntactically correct, the editor again examines DIRFLG. In earlier processing, the most significant bit ($80) of this byte was set on if the line was a direct statement. If it is not a direct statement, then the editor is finished with the line, and it waits for another input line. If the line is a direct statement, earlier processing already assigned it a line number of 32768 ($8000), one larger than the largest line number a user can enter. Since lines are arranged in the Statement Table in ascending numerical order, this line will have been inserted at the end of the table. The current statement pointer (STMCUR-$8A $8B) points to this line. The Program Editor transfers control to a Program Executor routine, Execution Control (EXECNL at location $A95F), which will handle the execution of the direct statement. (See Chapter 6.) 32

Part V

Pre-Compiler

5 Pre-Compiler

The symbols and symbol-combining rules of Atari BASIC are coded into Syntax Tables, which direct the Program Pre— compiler in examining source code and producing tokens. The information in the Syntax Tables is a transcription of a meta— language definition of Atari BASIC.

The Atari BASIC Meta-language A meta-language is a language which describes or defines another language. Since a meta-language is itself a language, it also has symbols and symbol-combining rules - which define with precision the symbols and symbol-combining rules of the subject language. Atari BASIC is precisely defined with a specially developed meta-language called the Atari BASIC Meta-language, or ABMI,. (ABML was derived from a commonly used compiler- technology meta-language called BNF.) The symbols and symbol-combining rules of ABML were intentionally kept very simple.

Making Up a Language To show you how ABML works, we'll create an extremely simple language called SAP, for Simple Arithmetic Process. SAP symbols consist of variables, constants, and operators.

- Variables: The letters A, B, and C only.
- Constants: The numbers 1,2,3,4,5,6,7,8, and 9 only.
- Operators: The characters +, -, *, /, and ! only. Of course, you already know the functions of all the operators except "!" The character! is a pseudo-operator of the SAP language used to denote the end of the expression, like the period that ends this sentence.

The grammar of the SAP language is precisely defined by the ABML definition in Figure 5-1. 33

Figure 5-1 The SAP Language Expressed in ABML

```
          SAP := <expression>!
 <expression> := <value> <operation> |
  <operation> := <operator> <expression>
      <value> := <constant> | <variable>
   <constant> := 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
   <variable> := A | B | C
   <operator> := + | - | * | /
```

The ABML symbols used to define the SAP language in Figure 5-1 are:

:=	is defined as	Whatever is on the left of : = is defined as consisting of whatever is on the right of :=, and in that order.
|	or	The symbol | allows choices for what something is defined as. For instance, in the sixth line <variable> can be A or B or C. If does not appear between two symbols, then there is no choice. For example, in the second line <expression> must have both <value> and <operation>, in that order, to be valid.
<>	label	Whatever comes between < and> is an ABML label. All labels, as non-terminal symbols, must be defined at some point, though the definitions can be circular - notice that <operation> is part of the definition of <expression> in the second line, while in the third line <expression> is part of the definition of <operation>.
	terminal	Symbols used in definitions, which are not

symbols — enclosed by < and > and are also not one of the ABML symbols, are terminal symbols in the language being defined by ABML. In SAP, some terminal symbols are A, !, B, *, and 1. They cannot be defined as consisting of other symbols - they are themselves the symbols that the SAP language manipulates, and must appear exactly as they are shown to be valid in SAP. In effect, they are the vocabulary of the SAP language.

Figure 5-2. The Generation of One Possible SAP Statement

```
(1)   SAP :=     <expression>!
(2)   SAP :=     <value> <operation>!
(3)   SAP :=     <variable> <operation>!
(4)   SAP :=     B< operation>!
(5)   SAP :=     B<operator> <expression>!
(6)   SAP :=     B*<expression>!
(7)   SAP :=     B*<value> <operation>!
(8)   SAP :=     B*<constant> <operation>!
(9)   SAP :=     B*4<operation>!
(10)  SAP :=     B*4< operator> <expression>!
(11)  SAP :=     B*4+<expression>!
(12)  SAP :=     8*4+<value> <operation>!
(13)  SAP :=     B*4+<variable> <operation>!
(14)  SAP :=     B*4+C<operation>!
(15)  SAP :=     B*4+C!
```

In (2), <value> <operation> replaces <expression> because the ABML definition of SAP (FigureS-I) defines <expression> as <value> <operation>.
In (3), the non-terminal <value> is replaced with <variable>. The definition of <value> gives two choices for the substitution of <value>. We happened to choose <variable>.
In (4), we reach a terminal symbol, and the process of defining <value> ends. We happened to choose **B** to replace <variable>.
In (5), we go back and start defining <operation>. There are two choices for the replacement of <operation>, either <operator> <expression> or nothing at all (since there is nothing to the right of in the second line of Figures 5-1). If nothing had been chosen, then (5) would have been: SAP :=B! The statement B! has no further non-terminals; the process would have been finished, and a valid statement would have been produced. Instead we happened to choose <operator> <expression>.

The SAP definition for <expression> is
<value> <operation>. If we replace <operation> with its definition we get:

<expression> : = <value> <operator> <expression>

The definition of <expression> includes <expression> as part of its definition. If the <operator> <expression> choice were always made for <operation>, then the process of replacement would never stop. A SAP statement can be infinitely long by definition. The only thing which prevents us from always having an infinitely long SAP statement is that there is a second choice for the replacement of <operation>: nothing.

The replacements in (5) and (10) reflect the repetitive choices of defining <expression> in terms of itself. The choice in (15) reflects the nothing choice and thus finishes the replacement process.

Computerized Statement Generation If we slightly modify our procedure for generating statements, we will have a process that could be easily

programmed into a computer. Instead of arbitrarily replacing the definition of non- terminals, we can think of the non-terminal as a GOSUB. When we see <X> := <Y> <Z>, we can think of <Y> as being a subroutine-type procedure:

(a) Go to the line that has <Y> on the left side.
(b) Process the definition (right side) of <Y>.
(c) If while processing the definition of <Y>, other non- terminals are found, GOSUB to them. (d) If while processing the definition of <Y> we encounter a terminal, output the terminal symbol as the next symbol of the generated statement.
(e) When the definition of <Y> is finished, return to the place that <Y> was called from and continue.

Since ABML is structured so that it can be programmed, a fascinating exercise is to design a simple English sentence grammar with ABML, then write a BASIC program to generate valid English sentences at random. The randomness of the sentences would be derived by using the RND function to select from the definitions or choices. An example of such a grammar is shown in Figure 5-3. (The programming exercise is left to you.)

Figure 5-3. A Simple English Sentence Grammar in ABML

```
   SENTENCE := <subject> <adverb> <verb> <object>.
  <subject> := The <adjective> <noun>
     <verb> := eats | sleeps | drinks | talks | hugs
   <adverb> := quickly | silently | slowly | viciously |
               lovingly | sadly |
   <object> := at home | in the car | at the table | at
               school | <subject>
     <noun> := boy | girl | dog | programmer | computer
               | teacher
<adjective> := happy | sad | blue | light | round | smart
               | cool | nice |
```

Syntactical Analysis The process of examining a language statement for grammatical correctness is called syntactical analysis, or *syntaxing*. Statement verification is similar to statement generation. Instead of arbitrarily choosing which or definition to use, however, the choices are already made, and we must check to see whether the statement symbols are used in valid patterns. To do this, we must process through each or definition until we find a matching valid terminal symbol. The result of statement generation is a valid, grammatically correct statement, but the result of statement verification is a statement validity indication, which is a simple yes or no. Either the statement is grammatically correct or it is not.Failure occurs when some statement symbol cannot be matched with a valid terminal symbol under the rules of the grammar.

The Reporting System To use the *pass/fail* result of statement verification, we must build a reporting system into the non4erminal checking process. Whenever we, in effect, GOSUB to a non-terminal definition, that non-terminal definition must report its pass/fail status. A *fail* status is generated and returned by a non-terminal definition when it finds no matching terminal for the current statement symbol. If the current statement symbol is B and the <constant> definition in the SAP language is called, then <constant> would report a *fail* status to the routine that called it. A *pass* status is returned when a terminal symbol is found which matches the current statement symbol. If our current statement symbol had been 7 instead of B, then <constant> would have reported *pass*. Whenever such a match does occur, we return to the statement, and the next symbol to the right becomes the new current symbol for examination and verification.

Cycling Through the Definitions In SAP, the <constant> definition is

called from the <value> definition. If <constant> reports *fail*, then we examine the next or choice, which is <variable>. The current symbol is B, so <variable> reports *pass*. Since at least one of the *or* choices of <value> has reported pass, <value> will report *pass* to its caller. If both <constant> and <variable> had reported *fail*, then <value> would report *fail* to its caller. The caller of <value> is <expression>. If <value> reports *pass*, <operation> is called. If <operation> reports pass, then <expression> can report *pass* to its caller. If either <value> or <operation> reports *fail*, then <expression> must report *fail*, since there are no other *or* choices for <expression> The definition of <operation> contains a special *pass/fail* property. If either <operator> or <expression> reports *fail*, 38

Chapter Five then the *or* choice must be examined. In this case the *or* choice is *nothing*. The *or nothing* means something special: report *pass*, but do not advance to the next symbol. The final *pass/fail* report is generated from the first line of the definition. If <expression> reports *pass* and the next symbol is!, then SAP reports *pass*. If either one of these conditions has a *fail* status, then SAP must report *fail* to whatever called SAP from outside the language.

Backing Up Sometimes it is necessary to back up over symbols which have already been processed. Let's assume that there was a definition of the type <X> : <Y>I<z>. It is possible that while <Y> is attempting to complete its definition, it will find a number of valid matching terminal symbols before it discovers a symbol that it cannot match. In this case, <Y> would have consumed a number of symbols before it decided to report *fail*. All of the symbols that <Y> consumed must be unconsumed before <Z> can be called, since <Z> will need to check those same symbols. The process of unconsuming symbols is called backup. Backup is usually performed by the caller of <Y>, which remembers which source symbol was current when it called <Y>. If <Y> reports *fail*, then the caller of <Y> restores the current symbol pointer before calling <Z>.

Locating Syntax Error When a final report of *fail* is given for a statement, it is often possible to guess where the error occurred. In a left-to-right system, the symbol causing the failure is usually the symbol which follows the rightmost symbol found to be valid. If we keep track of the rightmost valid symbol during the various backups, we can report a best guess as to where the failure- causing error is located. This is exactly what Atari BASIC does with the inverse video character in the ERROR line. For simplicity, our example was coded for SAP, but the syntactical analysis we have just described is essentially the process that the Atari BASIC pre-compiler uses to verify the grammar of a source statement. The Syntax Tables are an ABML description of Atari BASIC. The pre-compiler, also known as the *syntaxer*, contains the routines which verify BASIC statements. 39

Chapter Five Statement Syntax Tables There is one entry in the Syntax Tables for each BASIC statement. Each statement entry in the Syntax Table is a transcription of an ABML definition of the grammar for that particular statement. The starting address of the table entry for a particular statement is pointed to by that statement's entry in the Statement Name Table. The data in the Syntax Tables is very much like a computer machine language. The pseudo-computer which executes this pseudo-machine language is the pre-compiler code. Like any machine language, the pseudo-machine language of the Syntax Tables has instructions and instruction operands. For example, an ABML non4erminal symbol is transcribed to a code which the pre-compiler executes as a type of "GOSUB and report *pass/fail*" command. Here are the pseudo-instruction codes in the Syntax Tables; each is one byte in length.

Absolute Non-Terminal Vector

Name: ANTV
Code: $00

This is one of the forms of the non-terminal GOSUB. It is followed by the address, minus 1, of the nonAerminal's definition within the Syntax Table. The address is two bytes long, with the least significant byte first.

External Subroutine Call
Name: ESRT
Code: $01

This instruction is a special type of terminal symbol checker. It is followed by the address, minus 1, of a 6502 machine language routine. The address is two bytes long, with the least significant byte first. The ESRT instruction is a *deus ex machina* - the "god from the machine" who solved everybody's problems at the end of classical Greek plays. There are some terminals whose definition in ABML would be very complex and require a great many instructions to describe. In these cases, we go outside the pseudo-machine language of the Syntax Tables and get help from 6502 machine language routines - the *deus ex machina* that quickly gives the desired result. A numeric constant is one example of where this outside help is required.

ABML or
Name: OR
Value: $02

This is the familiar ABML or symbol (**|**). It provides for an alternative definition of a non-terminal.

Return
Name: RTN
Value: $03

This code signals the end of an ABML definition line. When we write an ABML statement on paper, the end of a definition line is obvious - there is no further writing on the line. When ABML is transcribed to machine codes, the definitions are all pushed up against each other. Since the function that is performed at the end of a definition is a return, the end of definition is called return (RTN).

Unused (Codes $04 through $0D are unused.)

Expression Non-Terminal Vector
Name: VFXP
Value: $0E

The ABML definition for an Atari BASIC expression is located at $A60D. Nearly every BASIC statement definition contains the possibility of having <expression> as part of it. VEXP is a single-byte call to <expression>, to avoid wasting the two extra bytes that ANTV would take. The pseudo- machine understands that this instruction is the same as an ANTV call to <expression> at $A60D.

Change Last Token
Name: CHNG
Value: $0F

This instruction is followed by a one-byte *change to* token value. The operator token instructions cause a token to be placed into the output buffer. Sometimes it is necessary to change the token that was just produced. For example, there are several = operators. One = operator is for the *assignment* statement LET X = 4. Another = operator is for

comparison operations like IF Y = 5. The pseudo-machine will generate the *assignment* = token when it matches =. The context of the grammar at that point may have required a *comparison* = token. The CHNG instruction rectifies this problem.

Operator Token
Name: (many)
Value: $10 through $7F

These instructions are terminal codes for the Atari BASIC Operators. The code values are the values of each operator token. The values, value names, and operator symbols are defined in the Operator Name Table (see Chapter 2). When the pseudo-machine sees these terminal symbol representations, it compares the symbol it represents to the current symbol in the source statement. If the symbols do not match, then fail status is generated. If the symbols match, then pass status is generated, the token (instruction value) is placed in the token output buffer, and the next statement source symbol becomes the current symbol for verification.

Relative Non-Terminal Vectors
Name: (none)
Value: $80- $BF (Plus) $C0 - $FF (Minus)

This instruction is similar to ANTV, except that it is a single byte. The upper bit is enough to signal that this one-byte code is a non-terminal GOSUB. The destination address of the GOSUB is given as a position relative to the current table location. The values $80 through $BF correspond to an address which is at the current table address plus $00 through $31'. The values $C0 through $FF correspond to an address which is at the current table address minus $01 through $31'.

Pre-compiler Main Code Description The pre-compiler, which starts at SYNENT ($A1C3), uses the pseudo-instructions in the Syntax Tables to verify the correctness of the source line and to generate the tokenized statements. The pre-compiler uses a LIFO stack in its processing. Each time a non-terminal vector ("GOSUB") is executed, the pre-compiler must remember where the call was made from. It must also remember the current locations in the input buffer (source statement) and the output buffer (tokenized statement) in case the called routine reports fail and backup is required. This LIFO stack is called the Syntax Stack. The Syntax Stack starts at 5480 at the label SIX. The stack is 256 bytes in size. Each entry in the stack is four bytes long. The stack can hold 64 levels of non4erminal calls. If a sixty-fifth stack entry is attempted, the LINE TOO LONG error is reported. (This error should be called LINE TOO COMPLEX, but the line is most likely too long also.) The first byte of each stack entry is the current input index (CIX). The second byte is the current output index (COX). The final two bytes are the current address within the syntax tables. The current stack level is managed by the STKLVL ($A9) cell. STKLVL maintains a value from $00 to $FC, which is the displacement to the current top of the stack entry.

Initialization The editor has saved an address in SRCADR ($96). This address is the address, minus 1, of the current statement's ABML instructions in the Syntax Tables. The current input index (CIX) and the current output index (COX) are also preset by the editor. The initialization code resets the syntax stack manager (STKLVL) to zero and loads the first stack entry with the values in CIX, COX, and CPC - the current program counter, which holds the address of the next pseudo-instruction in the Syntax Tables.

PUSH Values are placed on the stack by the PUSH routine ($A228). PUSH is entered with the new current pseudo-program counter value on the CPU stack. PUSH saves the current CIX, COX, and CPC on the syntax stack and

increments STKLVL. Next, it sets a new CPC value from the data on the CPU stack. Finally, PUSH goes to NEXT.

POP Values are removed from the stack with the POP routine ($A252). POP is entered with the 6502 carry flag indicating *pass/fail*. If the carry is clear, then pass is indicated. If the carry is set, then *fail* is indicated. POP first checks STKLVL. If the current value is zero, then the pre-compiler is done. In this case, POP returns to the editor via RTS. The carry bit status informs the editor of the *pass/fail* status. If STKLVL is not zero, POP decrements STKLVL. At this point, POP examines the carry bit status. If the carry is clear (*pass*), POP goes to NEXT. If the carry is set (*fail*), POP goes to FAIL.

NEXT and the Processes It Calls After initialization is finished and after each Syntax Table instruction is processed, NEXT is entered to process the next syntax instruction. NEXT starts by calling NXSC to increment CPC and get the next syntax instruction into the A register. The instruction value is then tested to determine which syntax instruction code it is and where to go to process it. If the Syntax Instruction is OR ($02) or RTN ($03), then exit is via POP. When POP is called due to these two instructions, the carry bit is always clear, indicating pass.

ERNTV. If the instruction is RNTV ("GOSUB" $80-$FF), then ERNTV ($A201) is entered. This code calculates the new CPC value, then exits via PUSH.

GETADR. If the instruction is ANTV ($00) or the *deus ex machina* ESRT ($01) instruction, then GETADR is called. GETADR obtains the following two-byte address from the Syntax Table. If the instruction was ANTV, then GETADR exits via PUSH. If the instruction was ESRT, then GETADR calls the external routine indicated. The external routine will report *pass/fail* via the carry bit. The *pass/fail* condition is examined at $A1FO. If *pass* is indicated, then NEXT is entered. If *fail* is indicated, then FAIL is entered.

TERMTST. If the instruction is VEXP ($0E), then the code at $A1F9 will go to TERMTST ($A2A9), which will cause the code at $A2AF to be executed for VEXP. This code obtains the address, minus 1, of the ABML for the <expression> in the Syntax Table and exits via PUSH.

ECHNG. If the instruction was CHNG ($0F), then ECHNG ($A2BA) is entered via tests at $A1F9 and $A2AB. ECHNG will increment CPC and obtain the *change-to* token which will then replace the last previously generated token in OUTBUFF. ECHNG exits via RTS, which will take control back to NEXT.

SRCONT. The Operator Token Instructions ($10-$7F) are handled by the SRCONT routine. SRCONT is called via tests at $A1F9 and $A2AD. SRCONT will examine the current source symbol to see if it matches the symbol represented by the operator token. When SRCONT has made its determination, it will return to the code at $A1FC. This code will examine the pass/fail (carry cleariset) indicator returned by SRCONT and take the appropriate action. (The SRCONT routine is detailed on the next page.)

FAIL If any routine returns a *fail* indicator, the FAIL code at $A26C will be entered. FAIL will sequentially examine the instructions, starting at the Syntax Table address pointed to by CPC, looking for an OR instruction. If an OR instruction is found, the code at $A27D will be entered. This code first determines if the current statement symbol is the rightmost source symbol to be examined thus far. If it is, it will update MAXCIX. The editor will use MAXCIX to set the inverse video flag if the statement is erroneous. Second, the code restores CIX and COX to their before-failure values and goes to NEXT to try this new OR choice.

If, while searching for an OR instruction, FAIL finds a RTN instruction, it will call POP with the carry set. Since the carry is set, POP will re-enter FAIL once it has restored things to the previous calling level. All instruction codes other than OR and RTN are skipped over by FAIL.

Pre-compiler Subroutine Descriptions SRCONT ($A2E6) The SRCONT code will be entered when an operator token instruction is found in the Syntax Tables by the main pre— compiler code. The purpose of the routine is to determine if the current source symbol in the user's line matches the terminal symbol represented by the operator token. If the symbols match, the token is placed into the output buffer and pass is returned. If the symbols do not match, fail is returned. SRCONT uses the value of the operator token to access the terminal symbol name in the Operator Name Table. The characters in the source symbol are compared to the characters in the terminal symbol. If all the characters match, *pass* is indicated.

TNVAR, TSVAR ($A32A) These *deus ex machina* routines are called by the ESRT instruction. The purpose of the routines is to determine if the current source symbol is a valid numeric (TNVAR) or string (TSVAR) variable. If the source symbol is not a valid variable, *fail* is returned. When *pass* is indicated, the routine will put a variable token into the output buffer. The variable token ($80-$FF) is an index into the Variable Name Table and the Variable Value Table, plus $80. The Variable Name Table is searched. If the variable is already in the table, the token value for the existing variable is used. If the variable is not in the table, it will be inserted into both tables and a new token value will be used. A source symbol is considered a valid variable if it starts with an alphabetic character and it is not a symbol in the Operator Name Table, which includes all the reserved words. The variable is considered to be a string if it ends with $; otherwise it is a numeric variable. If it is a string variable, $ is stored with the variable name characters. The routine also determines if the variable is an array by looking for (. If the variable is an array, (is stored with the variable name characters in the Variable Name Table. As a result, ABC, ABC$, and ABC(n) are all recognized as different variables.

TNCON ($A400) TNCON is called by the FSRT instruction. Its purpose is to examine the current source symbol for a numeric constant, using the floating point package. If the symbol is not a numeric constant, the routine returns fail. If the symbol is a numeric constant, the floating point package has converted it to a floating point number. The resulting six-byte constant is placed in the output buffer preceded by the $OE numeric constant token. The routine then exits with pass indicated.

TSCON ($A428) TSCON is called by the ESRT instruction. Its purpose is to examine the current symbol for a string constant. If the symbol is not a string constant, the routine returns *fail*. If the first character of the symbol is '', the symbol is a string constant. The routine will place the string constant token ($0F) into the output buffer, followed by a string length byte, followed by the string characters. The string constant consists of all the characters that follow the starting double quote up to the ending double quote. If the EOL character ($9B) is found before the ending double quote, an ending double quote is assumed. The EOL is not part of the string. The starting and ending double quotes are not saved with the string. All 256 character codes except $9B (EOL) and $22 (") are allowed in the string.

SEARCH ($A462) This is a general purpose table search routine used to find a source symbol character string in a table. The table to be searched is assumed to have entries which consist of a fixed length part (0 to 255 bytes) followed by a variable length ATASCII part. The last character of the ATASCII part is assumed to have the most significant

bit ($80) on. The last table entry is assumed to have the first ATASCII character as $00. Upon entry, the X register contains the length of the fixed part of the table (0 to 255). The A, Y register pair points to the start of the table to be searched. The source string for comparison is pointed to by INBUFF plus the value in CIX. Upon exit, the 6502 carry flag is clear if a match was found, and set if no match was found. The X register points to the end of the symbol, plus 1, in the buffer. The SRCADR ($95) two- byte cell points to the matched table entry. STENUM ($AF) contains the number, relative to zero, of the matched table entry.

SETCODE (A2C8) The SETCODE routine is used to place a token in the next available position in the output (token) buffer. The value in COX determines the current displacement into the token buffer. Mter the token is placed in the buffer, COX is incremented by one. If COX exceeds 255, the LINE TOO LONG error message is generated. 48

Part VI

Execution

6 Execution

During the editing and pre-compiling phase, the user's statements were checked for correct syntax, tokenized, and put into the Statement Table. Then direct statements were passed to the Program Executor for immediate processing, while program statements awaited later processing by the Program Executor. We now enter the execution phase of Atari BASIC. The Program Executor consists of three parts: routines which simulate the function of individual statement types; an expression execution routine which processes expressions (for example, A+B+3, A$(1,3), "HELP", A(3)+7.26E-13); and the Execution Control routine, which manages the whole process.

Execution Control Execution Control is invoked in two situations. If the user has entered a direct statement, Execution Control does some initial processing and then calls the appropriate statement execution routine to simulate the requested operation. If the user has entered RUN as a direct statement, the statement execution routine for RUN instructs Execution Control to start processing statements from the beginning of the statement table. When the editor has finished processing a direct statement, it initiates the Execution Control routine EXECNL ($A95F). Execution Control's job is to manage the process of statement simulation. The editor has saved the address of the statement it processed in STMCUR and has put the statement in the Statement Table. Since this is a direct statement, the line number is $8000, and the statement is saved as the last line in the Statement Table. The fact that a direct statement is always the last statement in the Statement Table gives a test for the end of a user's program. The high- order byte of the direct statement line number ($8000) has its most significant bit on. Loading this byte ($80) into the 6502 accumulator will set the minus flag on. The line number of any program statement isiess than; or equal to $7FFF. Loading the high order byte ($7F or less) of a program line number into the accumulator will set the 6502 minus flag off. This gives a simple test for a direct statement.

Initialization Execution Control uses several parameters to help it manage the task of statement execution. STMCUR holds the address in the Statement Table of the line currently being processed. LLNGTH holds the length of the current line. NXTSTD holds the displacement in the current line of the next statement to process. STMCUR already contains the correct value when Execution Control begins processing. SETLNI ($B81B) is called to store the correct values into LLNGTH and NXTSTD.

Statement Execution Since the user may have changed his or her mind about execution, the routine checks to see if the user hit the break key. If the user did hit BREAK, Execution Control carries out XSTOP ($B793), the same routine that is executed when the STOP statement is encountered. At the end of its execution, the XSTOP routine gives control to the beginning of the editor. If the user did not hit BREAK, Execution Control checks to see whether we are at the end of the tokenized line. Since this is the first statement in the line, we can't be at the end of the line. So why do the test? Because this part of the routine is executed once for each statement in the line in order to tell us when we do reach the end of the line. (The end-of-line procedure will be discussed later in this chapter.) The statement length byte (the displacement to the next statement in the line) is the first byte in a statement. (See Chapter 3.) The displacement *to* this byte was saved in NXTSTD. Execution Control now loads this new statement's displacement using the value in NXTSTD. The byte after the statement length in the line is the statement name token. Execution Control loads the statement name token into the A register. It saves the displacement to the next byte, the first of the statement's tokens, in STINDEX for the use of the statement simulation routines. The statement name token is used as an index to find this statement's entry in the Statement Execution Table.

Each table entry consists of the address, minus 1, of the routine that will simulate that statement. This simulation routine is called by pushing the address from the table onto the 6502 CPU stack and doing an RTS. Later, when a simulation routine is finished, it can do an RTS and return to Execution Control. (The name of most of the statement simulation routines in the BASIC listing is the statement name preceded by an X: XFOR, XRUN, XLIST.) Most of the statement simulation routines return to Execution Control after processing. Execution Control again tests for BREAK and checks for the end of the line. As long as we are not at end-of-line, it continues to execute statements. When we reach end-of-line, it does some end-of-line processing.

End-of-line Handling in a Direct Statement When we come to the end of the line in a direct statement, Execution Control has done its job and jumps to SNX3. The READY message is printed and control goes back to the Program Editor.

End-of-line Handling during Program Execution Program execution is initiated when the user types RUN. Execution Control handles RUN like any other direct statement. The statement simulation routine for RUN initializes STMCUR, NXTSTD, and LLNGTH to indicate the first statement of the first line in the Statement Table, then returns to Execution Control. Execution Control treats this first program statement as the next statement to be executed, picking up the statement name tokens and calling the simulation routines. Usually, Execution Control is unaware of whether it is processing a direct statement or a program statement. End-of- line is the only time the routine needs to make a distinction. At the end of every program line, Execution Control gets the length of the current line and calls GNXTL to update the address in STMCUR to make the next line in the Statement Table the new current line. Then it calls TENDST ($A9E2) to test the new line number to see if it is another program line or a direct statement. If it is a direct statement, we are at the end of the user's program. Since the direct statement includes the RUN command that started program execution, Execution Control does not execute the line. Instead, Execution Control calls the same routine that would have been calle'lif the program had contained an END statement (XEND, at$B78D). XEND does some end-of- program processing, causes READY to be printed, and returns to the beginning of the editor. If we are not at the end of the user's program, processing continues with the new current line.

Execution Control Subroutines TENDST ($A9E2) Exit parameters: The minus flag is set on if we are at the end of program. This routine checks for the end of the user's program in the Statement Table. The very last entry in the Statement Table is always a direct statement. Whenever the statement indicated by STMCUR is the direct statement, we have finished processing the user's program. The line number of a direct statement is $8000. The line number of any other statement is $7FFF or less. TENDST determines if the current statement is the direct statement by loading the high-order byte of the line number into the A register. This byte is at a displacement of one from the address in STMCUR. If this byte is $80 (a direct statement), loading it turns the 6502 minus flag on. Otherwise, the minus flag is turned off.

GETSTMT ($A9A2) Entry parameters: TSLNUM contains the line number of the statement whose address is required. Exit parameters: If the line number is found, the STMCUR contains the address of the statement and the carry flag is set off (clear). If the line number does not exist, STMCUR contains the address where a statement with that line number should be, and the carry flag is set on (set). The purpose of this routine is to find the address of the statement whose line number is contained in TSLNUM. The routine saves the address currently in STMCUR into SAVCUR and then sets STMCUR to indicate the top of the Statement Table. The line whose address is in STMCUR is called the current line or statement.

GETSTMT then searches the Statement Table for the statement whose line number is in TSLNUM. The line number in TSLNUM is compared to the line number of the current line. If they are equal, then the required statement has been found. Its address is in STMCUR, so GETSTMT clears the 6502 carry flag and is finished. If TSLNUM is smaller than the current statement line number, GETSTMT gets the length of the current statement by executing GETLL ($A9DD). GNXTL ($A9D0) is executed to make the next line in the statement table the current statement by putting its address into STMCUR. GETSTMT then repeats the comparison of TSLNUM and the line number of the current line in the same manner. If TSLNUM is greater than the current line number, then a line with this line number does not exist. STMCUR already points to where the line should be, the 6502 carry flag is already set, and the routine is done.

GETLL ($A9DD) Entry parameters: STMCUR indicates the line whose length is desired. Exit parameters: Register A contains the length of the current line. GETLL gets the length of the current line (that is, the line whose address is in STMCUR). The line length is at a displacement of two into the line. GETLL-loads the length into the A register and is done.

GNXTL ($A9D0) Entry parameters: STMCUR contains the address of the current line, and register A contains the length of the current line. Exit parameters: STMCUR contains the address of the next line. This routine gets the next line in the statement table and makes it the current line. GNXTL adds the length of the current line (contained in the A register) to the address of the current line in STMCUR. This process yields the address of the next line in the statement table, which replaces the value in STMCUR.

SETLN1 ($B81B) Entry parameters: STMCUR contains the address of the current line. Exit parameters: LLNGTH contains the length of the current line. NXTSTD contains the displacement in the line to the next statement to be executed (in this case, the first statement in the line). This routine initializes several line parameters so that Execution Control can process the line. The routine gets the length of the line, which is at a displacement of two from the start of the line. SETLN1 loads a value of three into the Y register to indicate the displacement into the line of the first statement and stores the value into NXTSTD as the displacement to the next statement for execution.

SETLINE ($B818) Entry parameters: TSLNUM contains the line number of a statement. Exit parameters: STMCUR contains the address of the statement whose line number is in TSLNUM. LLNGTH contains the length of the line. NXTSTD contains the displacement in the line to the next statement to be executed (in this case, the first statement in the line). Carry is set if the line number does not exist. This routine initializes several line parameters so that execution control can process the line. SETLINE first calls GETSTMT ($A9A2) to find the address of the line whose number is in TSLNUM and put that address into STMCUR. It then continues exactly like SETLN1.

Part VII

Execute Expression

7 Execute Expression

The Execute Expression routine is entered when the Program Executor needs to evaluate a BASIC expression within a statement. It is also the executor for the LET and implied LET statements. Expression operators have an order of precedence; some must be simulated before others. To properly evaluate an expression, Execute Expression rearranges it during the evaluation.

Expression Rearrangement Concepts Operator precedence rules in algebraic expressions are so simple and so unconscious that most people aren't aware of following them. When you evaluate a simple expression like Y=AX¯+BX+C, you don't think: "Exponentiation has a higher precedence than multiplication, which has a higher precedence than addition; therefore, I will first square the X, then perform the multiplication." You just do it. Computers don't develop habits or common sense - they have to be specifically commanded. It would be nice if we could just type Y = AX2+BX+C into our machine and have the computer understand, but instead we must separate all our variables with operators. We also have to learn a few new operators, such as * for multiply and ^ for exponentiation. Given that we are willing to adjust our thinking this much, we enter Y=A*X^2+B*X+C. The new form of expression does not quite have the same feel as Y AX2+BX+C; we have translated normal human patterns halfway into a form the computer can use. Even the operation X^2 causes another problem for the computer. It would really prefer that we give it the two values first, then tell it what to do with them. Since the computer still needs separators between items, we should write X^2 as X,2,^. Now we have something the computer can work with. It can obtain the two values X,2, apply the operator ^, and get a result without having to look ahead. If we were to transcribe X^2*A in the same manner, we would have X,2,^,A,*. The value returned by X,2,^ is the first value to multiply, so the value pair for multiplication is (X,2,^) and A. Again we have two values followed by an operator, and the computer can understand. If we continue to transcribe the expression by pairing values and operators, we find that we don't want to add the value X^2*A to B; we want to add the value X^2*A to B*X. Therefore, we need to tell the computer X,2,^,A,*,B,X,*, +. The value pair for the operator + is (X,2,^,A,*) and (B,X,*). The value pair for the final operation, =, is (X,2,^,A,*,B,X, *,+,C,+) and Y. So the complete translation of Y=AX2+BX+ C is X,2,^,A,*,B,X,*,+,C,+,Y,=. Very few people other than Forth programmers put up with this form of expression transcription. Therefore, Atari BASIC was designed to perform this translation for us, provided we use the correct symbols, like * and ^.

The Expression Rearrangement Algorithm The algorithm for expression rearrangement requires two LIFO stacks for temporary storage of the rearranged terms. The Operator Stack is used for temporarily saving operators; the Argument Stack is used for saving arguments. Arguments are values consisting of variables, constants, and the constant-like values resulting from previous expression operations.

Operator Precedence Table The *Atari BASIC User's Manual* lists the operators by precedence. The highest-precedence operators, like <, >, and = <, are at the top of the list; the lowest-precedence operator, OR, is at the bottom. The operators at the top of the list get executed before the operators at the bottom of the list. The operators in the precedence table are arranged in the same order as the Operator Name Table. Thus the token values can be used as direct indices to obtain an operator precedence value. The entry for each operator in the Operator Precedence Table contains two precedence values, the *go-onto-stack* precedence and the *come-off-stack* precedence. When a new operator has been plucked from an expression, its go-onto- stack precedence is tested in relation to the top-of-stack operator's come-off-stack precedence.

Expression Rearrangement Procedure The symbols of the expression (the arguments and the operators) are accessed sequentially from left to right, then rearranged into their correct order of precedence by the following procedure: 1. Initialize the Operator Stack with the Start Of Expression (SOE) operator. 2. Get the next symbol from the expression. 3. If the symbol is an argument (variable or constant), place the argument on the top of the Argument Stack. Go to step 2. 4. If the symbol is an operator, save the operator in the temporary save cell, SAVEOP. 5. Compare the go-onto-stack precedence of the operator in SAVEOP to the come-off stack precedence of the operator on the top of the Operator Stack. 6. If the top-of-stack operator's precedence is less than the precedence of the SAVEOP operator, then the SAVEOP operator is pushed onto the Operator Stack. When the push is done, go back to step 2. 7. If the top-of-stack operator's precedence is equal to or greater than the precedence of the SAVEOP operator, then pop the top-of-stack operator and execute it. When the execution is done, go back to step 5 and continue. The Expression Rearrangement Procedure has one apparent problem. It seems that there is no way to stop it. There are no exits for the "evaluation done" condition. This problem is handled by enclosing the expression with two special operators: the Start Of Expression (S0E) operator, and the End Of Expression (EOE) operator. Remember that SOE was the first operator placed on the Operator Stack, in step 1. Execution code for the SOE operator will cause the procedure to be exited in step 7, when SOE is popped and executed. The EOE operator is never executed. EOF's function is to force the execution of SOF. The precedence values of SOE and EOE are set to insure that SOE is executed only when the expression evaluation is finished The SOE come-off-stack precedence is set so that its value is always less than all the other operators' go-onto-stack precedence values. The EOE go-onto-stack precedence is set so that its value is always equal to or less than all the other operators' (including SOE's) corae-off-stack precedence values. Because SOE and EOE precedence are set this way, no operator other than EOE can cause SOE to be popped and executed. Second, EOE will cause all stacked operators, including SOE, to be popped and executed. Since SOE is always at the start of the expression and EOE is always at the end of the expression, SOE will not be executed until the expression is fully evaluated. In actual practice, the SOE operator is not physically part of the expression in the Statement Table. The Expression Rearrangement Procedure initializes the Operator Stack with the SOE operator before it begins to examine the expression. There is no single operator defined as the End Of Expression (EOE) operator. Every BASIC expression is followed by a symbol like :, THEN, or the EOL character. All of these symbols function as operators with precedence equivalent to the precedence of our phantom EOE operator. The THEN token, for example, serves a dual purpose. It not only indicates the THEN action, but also acts as the EOE operator when it follows an expression.

Expression Rearrangement Example To illustrate how the expression evaluation procedure works, including expression rearrangement, we will evaluate our Y = A*X^2 + B*X + C example and see how the expression is rearranged to X,2,^,A,*,B,X,*,+,C,+,Y,= with a correct result. To work our example, we need to establish a precedence table for the operators. The values in Figure 7-1 are similar to the actual values of these operators in Atari BASIC. The lowest precedence value is zero; the highest precedence value is $0F.

Example precedence Table operator go-on-stack come-off-stack symbol precedence precedence SOE NA $00 + $09 $09 * $0A $0A ^ s0C $0C = $0F $01 !(EOE) $00 NA 58

Chapter Seven Symbol values and notations. In the example steps, the term PS*n* refers to step *n* in the Expression Rearrangement Procedure (page 57). Step 5, for instance, will be called P55. In the actual

expression, the current symbol will be underlined. If B is the current symbol, then the actual expression will appear as Y=A*X 2+**B***X+C . In the rearranged expression, the symbols which have been evaluated up to that point will also be underlined. The values of the variables are: A=2 C=6 B=4 X=3 The variable values are assumed to be accessed when the variable arguments are popped for operator execution. The end-of-expression operator is represented by!.

Example step 1. Actual Expression: Y=A*X^2 + B*X + C! Rearranged Expression: X,2,^,A,*,B,X,*,+,C,+,Y,=,! Argument Stack: Operator Stack: SOE SAVEOP: PS1 has been executed. The Operator Stack has been initialized with the SOE operator. We are ready to start processing the expression symbols.

Example step 2. Actual Expression: **Y**= A*X^2+B*X+C! RearrangedExpression: X,2,^,A,*,B,X,*,+,C,+,Y,=,! Argument Stack: Y Operator Stack: SOE SAVEOP: The first symbol, Y, has been obtained and stacked in the Argument Stack according to PS2 and P53.

Example step 3. Actual Expression: Y**=**A*X^2+B*X+C! Rearranged Expression: X,2,^,A,*,B,X,*,+,C,+,Y,=,! Argument Stack: Y Operator Stack: SOE,= SAVEOP: = Operator = has been obtained via PS2. The relative precedences of SOE ($00) and = ($0F) dictate that the = be placed on the Operator Stack via PS6.

Example step 4. Actual Expression: Y=**A***X^2+B*X+C! Rearranged Expression: X,2,^,A,*,B,X,*,+,C,+,Y,=,! Argument Stack: Y,A Operator Stack: SOE, = SAVEOP: The next symbol is A. This symbol is pushed onto the Argument Stack via PS3.

Example step 5. Actual Expression: Y=A*****X^2+B*X+C! Rearranged Expression: X,2,^,A,*,B,X,*,+,C,+,Y,=,! Argument Stack: Y,A Operator Stack: SOE, = , * SAVEOP: * The next symbol is the operator *. The relative precedence of * and = dictates that * be pushed onto the Operator Stack.

Example step 6. Actual Expression: Y = A***X**^2+B*X+C! Rearranged Expression: X,2,^,A,*,B,X,*,+,C,+,Y,=,! Argument Stack: Y,A,X Operator Stack: SOE, = , * SAVEOP: The next symbol is the variable X. This symbol is stacked on the Argument Stack according to PS3.

Example step 7. Actual Expression: Y=A*X**^**2+B*X+C! Rearranged Expression: X,2,^,A,*,B,X,*,+C,+,Y,=,! Argument Stack: Y,A,X Operator Stack: SOE, = ,~, A U;' SAVEOP: A The next symbol is ^ The relative precedence of the and the * dictate that ^ be stacked via PS6.

Example step 8. Actual Expression: Y=A*X^**2**+B*X+C! Rearranged Expression X,2,^,A,*,B,X,*,+,C,+,Y,=,! Argument Stack: Y,A,X,2 Operator Stack: SOE,=,*,^ SAVEOP: The next symbol is 2. This symbol is stacked on the Argument Stack via PS3.

Example step 9. Actual Expression: Y=A*X^2**+**B*X+C! Rearranged Expression **X,2,^**,A,*,B,X,*,+,C,+,Y,=! Argument Stack: Y,A,9 Operator Stack: SOE,=,*, SAVEOP:+ The next symbol is the operator +. The precedence of the operator that was at the top of the stack, ^ , is greater than the precedence of +. PS7 dictates that the top-of-stack operator be popped and executed. The ^ operator is popped. Its execution causes arguments X and 2 to be popped from the Argument Stack, replacing the variable with the value that it represents and operating on the two values yielded: X^2=3^2=9. The resulting value, 9, is pushed onto the Argument Stack. The + operator remains in SAVEOP. We continue at PS5. Note that in the rearranged expression the first symbols, X,2,^, have been evaluated according to plan.

Example step 10. Actual Expression: Y=A*X^2**+**B*X+C! Rearranged Expression: **X,2,^,A**,*,B,X,*,+,C,+,Y,=,! Argument Stack: Y,18 Operator Stack: SOE,= SAVEOP: + This step originates at PS5. The SAVEOP operator, +, has a precedence that is less than the operator which was at the top of the stack, *. Therefore, according to P57, the is popped and executed. The execution of * results in A*9=2*9=18. The resulting value is pushed onto the Argument Stack.

Example step 11. Actual Expression: Y = A*X^2**+**B*X+C! Rearranged Expression: **X,2,^,A,*,**B,X,*,+,C,+,Y,=,! Argument Stack: Y,18 Operator Stack: SOE, =, + SAVEOP: When step 10 finished, we went to P55. The operator in SAVEOP was +. Since + has a higher precedence than the top-of-stack operator, =, the + operator was pushed onto the Operator Stack via PS6.

Example step 12. Actual Expression: Y=A*X^2+**B***X+C! Rearranged Expression: **X,2,^,A,*,**B,X,*,+,C,+,Y,=,! Argument Stack: Y,18,B Operator Stack: SOE, =, + SAVEOP: The next symbol is the variable B, which is pushed onto the Argument Stack via P53.

Example step 13. Actual Expression: Y = A*X^2+B*****X+C! Rearranged Expression: **X,2,^,A,***,B,X,*,+,C,+,Y,=,! Argument Stack: Y,18,B Operator Stack: SOE, =,+,* SAVEOP: * The next symbol is the operator *. Since * has a higher precedence than the top-of-stack +, * is pushed onto the stack via PS6.

Example step 14. Actual Expression: Y = A*X^2+B***X**+C! Rearranged Expression: **X,2,^,A,***,B,X,*,+,C,+,Y,=,! Argument Stack: Y,18,B,X Operator Stack: SOE, =, + * SAVEOP: The variable X is pushed onto the Argument Stack via PS3.

Example step 15. Actual Expression: Y=A*X^2+B*X**+**C! Rearranged Expression: **X,2,^,A,***,+,C,+,Y, =,! Argument Stack: Y,18,12 Operator Stack: SOE, =,+ SAVEOP: + The operator + is retneved from the expression. Since + has a lower precedence than which is at the top of the stack, * is popped and executed. The execution of * causes B*X=4*3=12. The resulting value of 12 is pushed onto the Argument Stack. We will continue at PS5 via the P57 exit rule.

Example step 16. Actual Expression: Y=A*X^2+B*X**+**C! Rearranged Expression: **X,2,^,A,*,B,X,*,+**,C,+,Y,=,! Argument Stack: Y,30 Operator Stack: SOE,= SAVEOP: + This step starts at PS5. The SAVEOP operator, +, has precedence that is equal to the precedence of the top-of-stack operator, also +. Therefore, + is popped from the operator stack and executed. The results of the execution cause 18+12, or 30, to be pushed onto the Argument Stack. PS5 is called. **Example step 17.** Actual Expression: Y=A*X^2+B*X**+**C! Rearranged Expression: **X,2,^,A,*,B,X,*,+**,C,+,Y,=,! Argument Stack: Y,30 Operator Stack: SOE, =,+ SAVEOP: This step starts at PS5. The SAVEOP is +. The top-of-stack operator, =, has a lower precedence than +; therefore, + is pushed onto the stack via PS6. **Example step 18.** Actual Expression: Y=A*X^2+B*X+**C**! Rearranged Expression: **X,2,^,A,*,B,X,*,+**,C,+,Y,=,! Argument Stack: Y,30,C Operator Stack: SOE, =,+ SAVEOP: The variable C is pushed onto the Argument Stack via PS3. 63

Chapter Seven Example step 19. Actual Expression: Y=A*X^2+B*X+C**!** Rearranged Expression: **X,2,^,A,*,B,X,*,+,C,+**,Y,=,! Argument Stack: Y,36 Operator Stack: SOE, SAVFOP: The EOE operator is plucked from the expression. The EOE has a lower precedence than the top-of-stack + operator. Therefore, + is popped and executed. The resulting value of 30+6, 36, is pushed onto the Argument Stack. PS5 will execute next.

Example step 20. Actual Expression: Y=A*X^2+B*X+C**!** Rearranged

Expression: **X,2,^,A,*,B,X,*,+,C,+,Y,=**,! Argument Stack: Operator Stack: SOE SAVEOP: ! This step starts at P55. The ! operator has a lower precedence than the top-of-stack = operator, which is popped and executed. The execution of = causes the value 36 to be assigned to Y. This leaves the Argument Stack empty. PS5 will be executed next.

Example step 21. Actual Expression: Y=A*X^2+B*X+C**!** Rearranged Expression: **X,2,^,A,*,B,X,*,+,C+,Y,=,!** Argument Stack: Operator Stack: SAVEOP: ! The operator in SAVEOP causes the SOE operator to be popped and executed. The execution of SOF terminates the expression evaluation. Note that the rearranged expression was executed exactly as predicted.

Mainline Code The Execute Expression code implements the Expression Rearrangement Procedure. The mainline code starts at the EXEXPR label at $AAE0. The input to EXEXPR starts at the current token in the current statement. STMCUR points to the current statement. STINDEX contains the displacement to the current token in the STMCUR statement. The output of EXEXPR is whatever values remain on the top of the argument stack when the expression evaluation is finished. In the following discussion, PSn refers to the procedure step *n* in the Expression Rearrangement Procedure. PS1, initialization, occurs when EXEXPR is entered. EXPINT is called to initialize the operator and argument stacks. EXPINT places the SOF operator on the operator stack. PS2, which obtains the next token, directly follows initialization at EXNXT ($AAE3). The code calls EGTOKEN to get the next expression symbol and classify it. If the token is an argument, the carry will be set. If the token is an operator, the carry will be clear. If the token is an argument, P53 is implemented via a call to ARGPUSH. After the argument is pushed onto the argument stack, EXNXT (PS2) will receive control. If the token was an operator, then the code at EXOT ($AAEE) will be executed This code implements P54 by saving the token in EXSVOP. PS5, which compares the precedents of the EXSVOP token and the top-of-stack token, follows EXOT at EXPTST ($AAFA). This code also executes the SOE operator If SOE is popped, then Execute Expression finishes via RTS. If the top-of-stack operator precedence is less than the EXSVOP operator precedence1 PS6 is implemented at EOPUSH ($AB15). EOPUSH pushes EXSVOP onto the operator stack and then goes to EXNXT (PS2). If the top-of-stack operator precedence is greater than or equal to the EXSVOP operator precedence, then PS7 is implemented at EXOPOP ($AB0B). EXOPOP will pop the top- of-stack operator and execute it by calling EXOP. When EXOP is done, control passes to EXPTST (PS5).

Expression Evaluation Stacks The two expression evaluation stacks, the Argument Stack and the Operator Stack, share a single 256-byte memory area. The Argument Stack grows upward from the lower end of the 256-byte area. The Operator Stack grows downward from the upper end of the 256-byte area. The 256-byte stack area is the multipurpose buffer at the start of the RAM tables. The buffer is pointed to by the ARGSTK (also ARGOPS) zero-page pointer at $80. The current index into the Argument Stack is maintained by ARSLVL ($AA). When the Argument Stack is empty, ARSLVL is zero. The OPSTKX cell maintains the current index into the Operator Stack. When the Operator Stack is initialized with the SOE operator, OPSTKX is inihalized to $FF. As operators are added to the Operator Stack, OPSTKX is decremented. As arguments are added to the Argument Stack, ARSLVL is incremented. Since the two stacks share a single 256-byte memory area, there is a possibility that the stacks will run into each other. The code at $ABC1 is used to detect a stack collision. It does this by comparing the values in ARSLVL and OPSTKX. If ARSLVL is greater than or equal to OPSTKX, then a stack collision occurs, sending the STACK OVERFLOW error to the user.

Operator Stack Each entry on the Operator Stack is a single-byte operator-type token. Operators are pushed onto the stack at EXOPUSH (SAB15) and are popped from the stack at EXOPOP ($ABOB).

Argument Stack Each entry on the Argument Stack is eight bytes long. The format of these entries is described in Figures 7-2, 7-3, and 7-4, and are the same as the formats for entries in the Variable Value Table. Unlike the Variable Value Table, the Argument Stack must deal with both variables and constants. In Figure 7-2, we see that VNUM is used to distinguish variable entries from constant entries. The SADR and AADR fields in the entries for strings and arrays are of special interest. (See Figures 7-3 and 7-4.) When a string or array variable is dimensioned, space for the variable is created in the string/array space. The displacement to the start of the variable's area within the string/array space is placed in the SADR/AADR fields at that time. A displacement is used rather than an absolute address because the absolute address can change if any program changes are made after the DIM statement is executed. Execute Expression needs these values to be absolute address values within the 6502 address space. When a string/array variable is retrieved from the Variable Value Table, the displacement is transformed to an absolute address. When (and if) the variable is put back into the Variable Value Table, the absolute address is converted back to a displacement. The entries for string constants also deserve some special attention. String constants are the quoted strings within the user program. These strings become part of the tokenized statements in the Statement Table. When Execute Expression gets a string token, it will create a string constant Argument Stack entry. This entry's SADR is an absolute address pointer to the string in the Statement Table. SLEN and SDIM are set to the actual length of the quoted string.

Argument Work Area An argument which is currently being examined by Execute Expression is kept in a special zero-page Argument Work Area (AWA). The AWA starts at the label VTYPE at $D2.

Figure 7-2. Argument Stack Entry

```
 0       1      2          8
 +-------+------+----------+
 | VTYPE | VNUM | DATA     |
 +-------+------+----------+
   |       |      |
   |       |      |
   |       |  Data Field. Format depends on VTYPE.
   |       |
   |       +----    If VNUM =0, the entry is a constant.
   |          If VNUM >0, the entry is a variable. In this case,
   |          the VNUM value is the entry number in the Variable
   |          Value Table. The token representing this variable is
   |          VNUM+$80.
   |
  +----     $00=Data is a six-byte floating point constant.
      $80=Data represents an undimensioned string.
      $81=Data represents a dimensioned string with
              a relative address pointer.
      $83=Data represents a dimensioned string with
              an absolute address pointer.
      $40=Data represents an undimensioned array.
      $41=Data represents a dimensioned array with
              a relative address pointer.
      $43=Data represents a dimensioned array with
              an absolute address pointer.
```

Figure 7-3. Argument Stack String Entry

```
      0       1      2      4      6      8
      +-------+------+------+------+------+
      | VTYPE | VNUM | SADR | SLEN | SDIM |
```

```
      +-------+------+------+------+------+
                     |      |      |
  +------------------+      |      |
  | +-----------------------+      |
  | | +----------------------------+
  | | |  Dimensioned length of the string. Valid only if
  | | +- VTYPE=$81 or $83.
  | |
  | +--- Current length of the string. Valid only if VTYPE
  |      =$81 or $83.
  |
  +----| String address. Valid only if VTYPE=$81 or $83.
       | If VTYPE=$81, then SADR is the displacement
       | of the start of the string from the start of the
       | string/array space.
       |
       | If VTYPE=$83, then SADR is the absolute address
       | of the start of the string.
```

Figure 7-4. Argument Stack Array Entry

```
      0       1      2      4      6      8
      +-------+------+------+------+------+
      | VTYPE | VNUM | SADR | DIM1 | DIM2 |
      +-------+------+------+------+------+
                     |      |      |
  +------------------+      |      |
  | +-----------------------+      |
  | | +----------------------------+
  | | |  When an array has been dimensioned as A(DI,D2),
  | | +- this field contains the D2 value. If an array
  | |    was dimensioned as A(DI), then this field is
  | |    zero. The field is valid only if VTYPE=$41 or
  | |    $43.
  | |
  | |    When an array has been dimensioned, as A(D1,D2)
  | |--- or as A(D1), this field contains the D1 value.
  |      The field is valid only if VTYPE=$41 or $43.
  |
  |    | Array Address. Valid only if VTYPE=$41 or $43.
  +----|
       | If VTYPE=$41, the AADR is the displacement to
       | the start of the array in the string/array space.
       |
       | If VTYPE=$43, the AADR is the absolute address
       | of the start of the string.
```

Array Address. Valid only if VTYPE=$41 or $43. +----| | If VTYPE=$41, the AADR is the displacement to | the start of the array in the string/array space. | | If VTYPE=$43, the AADR is the absolute address | of the start of the string. 68

Operator Executions An operaror is executed when it is popped from the Operator Stack. Execute Expression calls EXOP at $AB20 to start this execution. The EXOP routine uses the operator token value as an index into the Operator Execution Table ($AA70). The operator execution address from this table, minus 1, is placed on the 6502 CPU stack. An RTS is then executed to begin executing the operator's code. The names of the operator execution routines all begin with the characters XP. All the Atari BASIC functions, such as PEEK, RND, and ABS, are executed as operators. Most routines for the execution of the operators are very simple and straightforward. For example, the * operator routine, XPMUL ($AC96), pops two arguments, multiplies them via the floating point

package, pushes the result onto the argument stack, and returns.

String, Array, DIM, and Function Operations Any array reference in an expression may be found in one of two forms: A(x) or A(x,y). The indices x andy may be any valid expression. The intent of the indices is to reference a specific array element. Before the specific element reference can take place, the x and/or y index expressions must be fully evaluated. To do this, the characters '(' ',' and ')' are made operators. The precedence of these operators forces things to happen in the correct sequence. Figure 7-5 shows the relative precedence of these operators for an array.

Figure 7-5. Array Operator Precedence

operator symbol	go-on-stack precedence	come-off-stack precedence
(	$0F	$02
,(comma)	$04	$03
)	$04	$0E

As a result of these precedence values, (has a high enough precedence to go onto the stack, no matter what other operator is on the top of the stack.

The comma's go-on-stack precedence will force all operators except (to be popped and executed. As a result, the x index sub-expression in the expression A(x,y), will be fully evaluated and the final x index value will be pushed onto the Argument Stack.

The comma will then be placed onto the Operator Stack. Its come-off-stack precedence is such that no other operator, except), will pop it off.

The) operator precedence will force any y index expression to be fully evaluated and the y index result value to be placed onto the Argument Stack.

It will then force the comma operator to be popped and executed. This action results in a comma counter being incremented.

The) will then force the (to be popped and executed. The execution of (results in the proper array element being referenced. The (operator will pop the indices from the Argument Stack. The number of indices (either zero or one) to be popped is governed by the comma counter, which was incremented by one for each comma that was popped and executed.

Atari BASIC has numerous (tokens, and each causes a different (routine to be executed. These (operators are array (CALPRN), string (CSLPRN) array DIM (CDLPRN) string DIM (CDSLPR), function (CFLPRN), and the expression grouping CLPRN operator. The Syntax Table pseudo-instruction CHNG is used to change the CLPRN token to the other (tokens in accordance with the context of the grammar.

The expression operations for each of these various (operators in relation to commas and (is exactly the same. When (is executed, the comma count will show how many arguments the operator's code must pop from the argument stack. Each of these arguments will have been evaluated down to a single value in the form of a constant.

Part VIII

Execution Boundary Conditions

Converted 2006 by Andreas Bertelmann for ABBUC

8 Execution Boundary Conditions

BASIC Language statements can be divided into groups with related functions. The execution boundary statements, RUN, STOP, CONT and END, cause a BASIC program to start or stop executing. The routines which simulate these statements are XRUN, XSTOP, XCONT, and XEND.

Program Termination Routines Any BASIC statement can be used as either a direct statement or a program statement, but some only make sense in one mode. The STOP statement has no real meaning when entered as a direct statement. When the statement simulation routine for STOP is asked to execute in direct mode, it does as little processing as possible and exits. Useful processing occurs only when STOP is a program statement.

STOP ($B7A7). The XSTOP and XEND routines are similar and perform some of the same tasks. The tasks common to both are handled by the STOP routine. If this statement is not a direct statement, the STOP routine saves the line number of the current line in STOPLN. This line number is used later for printing the STOPed message. It is also used by the CONT simulation routine (XCONT) to determine where to restart program execution. (Since XEND also uses this routine, it is possible to CONTinue after an END statement in the middle of a program.) The STOP routine also resets the LIST and ENTER devices to the screen and the keyboard.

XSTOP ($B793). XSTOP does the common STOP processing and then calls :LPRTOKEN($B535) to print the STOPed message. It then calls one of the error printing routines, :ERRM2 ($B974), to output the AT LINE *nnn* portion. The :ERRM2 routine will not print anything if this was a direct statement. When :ERRM2 is finished, it jumps back to the start of the editor.

XEND ($B7SD). XEND calls the STOP routine to save the current line number. It then transfers to the start of the editor via the SNX1 entry point. This turns off the sound, closes any open IOCBs, and prints the READY message. XEND also leaves values on the 6502 CPU stack. These values are thrown away when the editor resets the stack.

END OF PROGRAM. A user may have neglected to include an END statement in his program. In this case, when Execution Control comes to the end of the Statement Table it calls XEND, and the program is terminated exactly as if the last statement in the program were an END.

Program Initiation Routines The statements that cause a user's program to begin execution are RUN and CONT. These statements are simulated by XRUN and XCONT

XCONT ($B7BE). The CONT statement has no meaning when encountered as a program statement, so its execution has no effect. When the user enters CONT as a direct statement, XCONT uses the line number that was saved in STOPLN to set Execution Control's line parameters (STMCUR, NXTSTD, and LLNGTH). This results in the current line being the line following the one whose line number is in STOPLN. This means that any statement following STOP or END on a line will not be executed; therefore, STOP and END should always be the last statement in the line. If we are at the end of the Statement Table, XCONT terminates as if an END statement had been encountered in the program. If there are more lines to process, XCONT returns to Execution Control, which resumes processing at the line whose address was just put into STMCUR.

XRUN ($B74D). The RUN statement comes in two formats, RUN and RUN <filespec>. In the case of RUN <filespec>, XRUN executes XLOAD to load a saved program, which replaces the current one in memory. The process

then proceeds like RUN. XRUN sets up Execution Control's line pointers to indicate the first line in the Statement Table. It clears some flags used to control various other BASIC statements; for example, it resets STOPLN to O. It closes all IOCBs and executes XCLR to reset all the variables to zero and get rid of any entries in the String/Array Table or the Runtime Stack. If there is no program, so the only thing in the Statement Table is the direct statement, then XRUN does some clean-up, prints READY, and returns to the start of the editor, which resets the 6502 CPU stack. If there is a program, XRUN returns to Execution Control, which starts processing the first statement in the table as the current statement. When RUN <filespec> is used as a program statement, it performs the useful function of chaining to a new program, but if RUN alone is used as a program statement, an infinite loop will probably result.

Error Handling Routine There are other conditions besides the execution boundary statements that terminate a program's execution. The most familiar are errors. There are two kinds of errors that can occur during execution: Input/Output errors and BASIC language errors. Any BASIC routine that does I/O calls the IOTEST routine ($BCB3) to check the outcome of the operation. If an error that needs to be reported to the user is indicated, IOTEST gets the error number that was returned by the Operating System and joins the Error Handling Routine, ERROR ($B940), which finishes processing the error. When a BASIC language error occurs, the error number is generated by the Error Handling Routine. This routine calculates the error by having an entry point for every BASIC language error. At each entry point, there is a 6502 instruction that increments the error number. By the time the main routine, ERROR, is reached, the error number has been generated. The Error Handling Routine calls STOP ($B7A7) to save the line number of the line causing the error in STOPLN. It tests TRAPLN to see if errors are being TRAPed. The TRAP option is on if TRAPLN contains a valid line number. In this case, the Error Handler does some clean-up and joins XGOTO, which transfers processing to the desired line. If the high-order byte of the line number is $80 (not a valid line number), then we are not TRAPing errors. In this case, the Error Handler prints the four-part error message, which consists of ERROR, the error number, AT LINE, and finally the line number. If the line in error was a direct statement, the AT LINE part is not printed. The error handler resets ERRNUM to zero and is finished. The Error Handling Routine does not do an orderly return, but jumps back to the start of the editor at the SYNTAX entry point where the 6502 stack is reset, clearing it of the now- unwanted return addresses.

Part IX

Program Flow Control Statements

9 Program Flow Control Statements

Execution Control always processes the statement in the Statement Table that follows the one it thinks it has just finished. This means that statements in a BASIC program are usually processed in sequential order. Several statements, however, can change that order: GOTO, IF, TRAP, FOR, NEXT, GOSUB, RETURN, POP, and ON. They trick Execution Control by changing the parameters that it maintains.

Simple Flow Control Statements XGOTO ($BGA3) The simplest form of flow control transfer is the GOTO statement, simulated by the XGOTO routine. Following the GOTO token in the tokenized line is an expression representing the line number of the statement that the user wishes to execute next. The first thing the XGOTO routine does is ask Execute Expression to evaluate the expression and convert it to a positive integer. XGOTO then calls the GETSTMT routine to find this line number in the Statement Table and change Execution Control's line parameters to indicate this line. If the line number does not exist, XGOTO restores the line parameters to indicate the line containing the original GOTO, and transfers to the Error Handling Routine via the ERNOLN entry point. The Error Handling Routine processes the error and jumps to the start of the editor. If the line number was found, XGOTO jumps to the beginning of Execution Control (EXECNL) rather than returning to the point in the routine from which it was called. This leaves garbage on the 6502 CPU stack, so XGOTO first pulls the return address off the stack.

XIF($8778) The IF statement changes the statement flow based on a condition. The simulatiQn routine, xw, begins by calling a subroutine of Execute Expressionto evaluate the condition. Since this is a logical (rather than an arithmetic) operation, we are only interested in whether the value is zero or non-zero. If the expression was false (non-zero), XIF modifies Execution Control's line parameters to indicate the end of this line and then returns. Execution Control moves to the next line, skipping any remaining statements on the original IF statement line. If the expression is true (zero), things get a little more complicated. Back during syntaxing, when a statement of the form IF <expression> THEN <statement> was encountered, the pre-compiler generated an end-of-statement token after THEN. XIF now tests for this token. If we are at the end of the statement, XIF returns to Execution Control, which processes what it thinks is the next statement in the current line, but which is actually the THEN <statement> part of the IF statement. If XIF does not find the end-of-statement token, then the statement must have had the form IF <expression> THEN <line number>. XIF jumps to XGOTO, which finishes processing by changing Execution Control's line parameters to indicate the new line.

XTRAP ($B7E1) The TRAP statement does not actually change the program flow when it is executed. Instead, the XTRAP simulation routine calls a subroutine of Execute Expression to evaluate the line number and then saves the result in TRAPLN ($BC). The program flow is changed only if there is an error. The Error Handling Routine checks TRAPLN. If it contains a valid line number, the error routine does some initial set-up and joins the XGOTO routine to transfer to the new line.

Runtime Stack Routines The rest of the Program Flow Control Statements use the Runtime Stack. They put items on the stack, inspect them, and/or remove them from the stack. Every item on the Runtime Stack contains a four-byte header. This header consists of a one-byte type indication, a two-byte line number, and a one-byte displacement to the Statement Name Token. (See pages 18-19.) The type byte is the last byte placed on the stack for each entry. This means that the pointer to the top of the Runtime Stack (RUNSTK) points to the type byte of the most recent entry on the stack. A zero type byte indicates a GOSUB-type entry. Any non-zero type byte represents a FOR-type entry A GOSUB entry consists solely

of the four-byte header. A FOR entry contains twelve additional bytes: a six-byte limit value and a six-byte step value. Several routines are used by more than one of the statement simulation routines.

PSHRSTK ($B683) This routine expands the Runtime Stack by calling EXPLOW and then storing the type byte, line number, and displacement of the Statement Name Token on the stack.

POPRSTK ($B841) This routine makes sure there really is an entry on the Runtime Stack. POPRSTK saves the displacement to the statement name token in SVI)ISP, saves the line number in TSLNUM, and puts the type/variable number in the 6502 accumulator. It then removes the entry by calling the CONTLOW routine.

:GETTOK ($B737) This routine first sets up Execution Control's line parameters to point to the line whose number is in the entry just pulled from the Runtime Stack. If the line was found, : GETTOK updates the line parameters to indicate that the statement causing this entry is now the current statement. Finally, it loads the 6502 accumulator with the statement name token from the statement that created this entry and returns to its caller. If the line number does not exist, : GETTOK restores the current statement address and exits via the ERGFDEL entry point in the Error Handling Routine. Now let's look at the simulation routines for the statements that utilize the Runtime Stack.

XFOR ($B64B) XFOR is the name of the simulation routine which executes a FOR statement. In the statement FOR 1=1 TO 10 STEP 2: *I* is the *loop control variable* 1 is its *initial value* 10 is the *limit value* 2 is the *step value* XFOR calls Execute Expression, which evaluates the initial value and puts it in the loop control variable's entry in the Variable Value Table. Then it calls a routine to remove any currently unwanted stack entries for example, a previous FOR statement that used the same loop control variable as this one. XFOR calls a subroutine of Execute Expression to evaluate the limit and step values. If no step value was given, a value of 1 is assigned. It expands the Runtime Stack using EXPLOW and puts the values on the stack. XFOR uses PSHRSTK to put the header entry on the stack. It uses the variable number ofthe loop control variable (machine-language ORed with $80) as the type byte. XFOR now returns to Execution Control, which processes the statement following the FOR statement. The FOR statement does not change program flow. It just sets up an entry on the Runtime Stack so that the NEXT statement can change the flow.

XNEXT ($B6CF) The XNEXT routine decides whether to alter the program flow, depending on the top Runtime Stack entry. XNEXT calls the POPRSTK routine repeatedly to remove four-byte header entries from the top of the stack until an entry is found whose variable number (type) matches the NEXT statement's variable token. If the top-of-stack or GOSUB-type entry is encountered, XNEXT transfers control to an Error Handling Routine via the ERNOFOR entry point. To compute the new value of the loop variable, XNEXT calls a subroutine of Execute Expression to retrieve the loop control variable's current value from the Variable Value Table, then gets the step value from the Runtime Stack, and finally adds the step value to the variable value. XNEXT again calls an Execute Expression subroutine to update the variable's value in the Variable Value Table. XNEXT gets the limit value from the stack to determine if the variable's value is at or past the limit. If so, XNEXT returns to Execution Control without changing the program flow, and the next sequential statement is processed. If the variable's value has not reached the limit, XNEXT returns the entry to the Runtime Stack and changes the program flow. POPRSTK already saved the line number of the FOR statement in TSLNUM and the displacement to the statement name token in SVDISP. XNEXT calls the GETTOK routine to indicate the FOR statement as the current statement. If the token at the saved displacement is not

a FOR statement name token, then the Error Handling Routine is given control at the ERGFDEL entry point. Otherwise, XNEXT returns to Execution Control, which starts processing with the statement following the FOR statement.

XGOSUB ($BGA0) The GOSUB statement causes an entry to be made on the Runtime Stack and also changes program flow. The XGOSUB routine puts the GOSUB type indicator (zero) into the 6502 accumulator and calls PSHRSTK to put a four-byte header entry on the Runtime Stack for later use by the simulation routine for RETURN. XGOSUB then processes exactly like XGOTO.

XRTN ($B719) The RETURN statement causes an entry to be removed from the Runtime Stack. The XRTN routine uses the information in this entry to determine what statement should be processed next. The XRTN first calls POPRSTK to remove a GOSUB-type entry from the Runtime Stack. If there are no GOSUB entries on the stack, then the Error Handling Routine is called at ERBRTN. Otherwise, XRTN calls GETTOK to indicate that the statement which created the Runtime Stack entry is now the current statement. If the statement name token at the saved displacement is not the correct type, then XRTN exits via the Error Handling Routine's ERGFDEL entry point. Otherwise, control is returned to the caller. When Execution Control was the caller, then GOSUB must have created the stack entry, and processing will start at the statement following the GOSUB. Several other statements put a GOSUB-type entry on the stack when they need to mark their place in the program. They do not affect program flow and will be discussed in later chapters.

XPOP ($B841) The XPOP routine uses POPRSTK to remove an entry from the Runtime Stack. A user might want to do this il he decided not to RETURN from a GOSUB.

XON ($B7ED) The ON statement comes in two versions: ON-GOTO and ON-GOSUB. Only ON-GOSUB uses the Runtime Stack. The XON routine evaluates the variable and converts it to an integer (MOD 256). If the value is zero, XON returns to Execution Control without changing the program flow. If the value is non-zero and this is an ON-GOSUB statement, XON puts a GOSUB-type entry on the Runtime Stack for RETURN to use later. From this point, ON-GOSUB and ON-GOTO perform in exactly the same manner. XON uses the integer value calculated earlier to index into the tokenized statement line to the correct GOTO or GOSUB line number. If there is no line number corresponding to the index, XON returns to Execution Control without changing program flow. Otherwise, XON joins XGOTO to finish processing.

Part X

Tokenized Program Save and Load

10 Tokenized Program Save and Load

The tokenized program can be saved to and reloaded from a peripheral device, such as a disk or a cassette. The primary statement for saving the tokenized program is SAVE. The saved program is reloaded into RAM with the LOAD statement. The CSAVE and the CLOAD statements are special versions of SAVE and LOAD for use with a cassette.

Saved File Format The tokenized program is completely contained within the Variable Name Table, the Variable Value Table, and the Statement Table. However, since these tables vary in size, we must also save some information about the size of the tables. The SAVE file format is shown in Figure 10-1. The first part consists of seven fields, each of them two bytes long, which tell where each table starts or ends. Part two contains the saved program's Variable Name Table (VNT), Variable Value Table (VVT), and Statement Table (ST). The displacement value in all the part-one fields is actually the displacement p?us 256. We must subtract 256 from each displacement value to obtain the true displacement. The VNT starts at relative byte zero in the file's second part. The second field in part one holds that value plus 256. The DVVT field in part one contains the displacement, minus 256, of the VVT from the start of part two. The DST value, minus 256, gives the displacement of the Statement Table from the start of part two. The DEND value, minus 256, gives the end-of-file displacement from the start of part two.

Figure 10-1. SAVE File Format

```
PART 1     0 +----------+
       |     0    |
     2 +----------+
       |    256   | <--> The displacement of the VNT from
     4 +----------+      the beginning of part two, plus 256.
       | Not Used |
     6 +----------+
       |   DVVT   | <--> The displacement of VNT from the
     8 +----------+      beginning of part two, plus 256.
       |    DST   | <--> The displacement of ST from the
     10+----------+      beginning of part two, plus 256.
       | Not Used |
     12+----------+
       |   DEND   | <--> The displacement of the end of the
======14+==========+      file from the beginning of part two.
PART 2 0|   VNT    | <--> Variable Name Table
DVVT-256+----------+
        |   VVT    | <--> Variable Value Table
DSNT-256+----------+
        |    ST    | <--> Statement Table
DEND-256+----------+
```

XSAVE($BB5D) The code that implements the SAVE statement starts at the XSAVE ($BB5D) label. Its first task is to open the specified output file, which it does by calling ELADVC. The next operation is to move the first seven RAM table pointers from $80 to a temporary area at $500. While these pointers are being moved, the value contained in the first pointer is subtracted from the value in each of the seven pointers, including the first. Since the first pointer held the absolute address of the first RAM table, this results in a list of displacements from the first RAM table to each of the other tables. These seven two-byte displacements are then written from the temporary area to the file via 103. These are the first fourteen bytes of the SAVE file. (See Figure 10-1.) The first RAM table is the 256-byte buffer, which will not be SAVEd. This is why the seven two-byte fields at the beginning of the SAVEd file hold values exactly 256 more than the true displacement of the tables they point to. (The LOAD procedure will resolve the 256-byte

discrepancy.) The next operation is to write the three needed RAM tables. The total length of these tables is determined from the value in the seventh entry in the displacement list, minus 256. To write the three entries, we point to the start of the Variable Name Table and call 104, with the length of the three tables. This saves the second part of the file format. The file is then closed and XSAVE returns to Execution Control.

XLOAD ($BAFB) The LOAD statement is implemented at the XLOAD label located at $BAFB. XLOAD first opens the specified load file for input by calling ELADVC. BASIC reads the first fourteen bytes from the file into a temporary area starting at $500. These fourteen bytes are the seven RAM table displacements created by SAVE. The first two bytes will always be zero, according to the SAVE file format. (See Figure 10-1.) BASIC tests these two bytes for zero values. If these bytes are not zero, BASIC assumes the file is not a valid SAVE file and exits via the ERRNSF, which generates error code 21 (Load File Error). If this is a valid SAVE file, the value in the pointer at $80 (Low Memory Address) is added to each of the seven displacements in the temporary area. These values will be the memory addresses of the three RAM tables, if and when they are read into memory. The seventh pointer in the temporary area contains the address where the end of the Statement Table will be. If this address exceeds the current system high memory value, the routine exits via ERRPTL, which generates error code 19 (Load Program Too Big). If the program will fit, the seven addresses are moved from the temporary area to the RAM table pointers at $80. The second part of the file is then loaded into the area now pointed to by the Variable Name Table pointer $82. The file is closed, CLR is executed, and a test for RUN is made. If RUN called XLOAD, then a value of $FF was pushed onto the CPU stack. If RUN did not call XLOAD, then $00 was pushed onto the CPU stack. If RUN was the caller, then an RTS is done. If XLOAD was entered as a result of a LOAD or CLOAD statement, then XLOAD exits directly to the Program Editor, not to Execution Control.

CSAVE and CLOAD The CSAVE and CLOAD statements are special forms of SAVE and LOAD. These two statements assume that the SAVE/LOAD device is the cassette device; CSAVE is not quite the same as SAVE "C:". Using SAVE with the "C:" device name will cause the program to be saved using long cassette inter-record gaps. This is a time waster, and CSAVE uses short inter-record gaps. CSAVE starts at XCSAVE ($BBAC). CLOAD starts at XCLOAD ($BBA4).

Part XI

The LIST and ENTER Statements

11 The LIST and ENTER Statements

LIST can be used to store a program on an external device and ENTER can retrieve it. The difference between LOAD-SAVE and LIST-ENTER is that LOAD-SAVE deals with the tokenized program, while LIST-ENTER deals with the program in its source (ATASCII) form.

The ENTER Statement BASIC is in ENTER mode whenever a program is not RUNning. By default the Program Editor looks for lines to be ENTERed from the keyboard, but the editor handles all ENTERed lines alike, whether they come from the keyboard or not.

The Enter Device To accomplish transparency of all input data (not just ENTERed lines), BASIC maintains an enter device indicator, ENTDTD ($B4). When a BASIC routine (for example, the INPUT simulation routine) needs data, an I/O operation is done to the IOCB specified in ENTDTD. When the value in ENTDTD is zero, indicating IOCB 0, input will come from the keyboard. When data is to come from some other device, ENTDTD contains a number indicating the corresponding IOCB. During coldstart initialization, the enter device is set to IOCB 0. It is also reset to 0 at various other times.

XENTER ($BACB) The XENTER routine is called by Execution Control to simulate the ENTER statement. XENTER opens IOCB 7 for input using the specified <filespec>, stores a 7 in the enter device ENTDTD, and then jumps to the start of the editor.

Entering from a Device When the Program Editor asks GLGO, the get line routine ($BA92), for the next line, GLGO tells CIO to get a line from the device spedified in ENTDTD - in this case, fr9m IOCB 7. The editor continues to process lines from IOCB 7 until an end-of- file error occurs. The IOTEST routine detects the EOF condition, sees thatwe are using IOCB 7 for ENTER, closes device 7, and jumps to SNX2 to reset the enter device (ENTDTD) to 0 and print the READY message before restarting at the beginning of the editor.

The LIST Statement The routine which simulates the LIST statement, XLIST, is actually another example of a language translator, complete with symbols and symbol-combining rules. XLIST translates the tokens generated by Atari BASIC back into the semi- English BASIC statements in ATASCII. This translation is a much simpler task than the one done by the pre-compiler, since XLIST can assume that the statement to be translated is syntactically correct. All that is required is to translate the tokens and insert blanks in the appropriate places.

The List Device BASIC maintains a list device indicator, LISTDTD ($B5), similar to the enter device indicator discussed earlier. When a BASIC routine wants to output some data (an error message, for example), the I/O operation is done to the device (IOCB) specified in LISTDTD. During coldstart initialization and at various other times, LISTDTD is set to zero, representing IOCB 0, the editor, which will place the output on the screen. Routines such as XPRINT or XLIST can change the LIST device to indicate some other IOCB. Thus the majority of the BASIC routines need not be concerned about the output's destination. Remember that IOCB 0 is always open to the editor, which gets input from the keyboard and outputs to the screen. IOCB 6 is the S: device, the direct access to graphics screen, which is used in GRAPHICS statements. Atari BASIC uses IOCB 7 for I/O commands that allow different devices, like SAVE, LOAD, ENTER, and LIST.

XLIST ($B483) The XLIST routine considers the output's destination in its initialization process and then forgets about it. It looks at the first expression in the tokenized line. If it is the <filespec> string, XLIST calls a routine to open the specified device using IOCB 7 and to

store a 7 in LISTDTD. All of XLIST's other processing is exactly the same, regardless of the LISTed data's final destination. XLIST marks its place in the Statement Table by calling a subroutine of XGOSUB to put a GOSUB type entry on the Runtime Stack. Then XLIST steps through the Statement Table in the same way that Execution Control does, using Execution Control's line parameters and subroutines. When XLIST is finished, Execution Control takes the entry off the Runtime Stack and continues. The XLIST routine, assuming it is to LIST all program statements, sets default starting and ending line numbers of 0 (in TSLNUM) and $7FFF (in LELNUM). XLIST then determines whether line numbers were specified in the tokenized line that contained the LIST statement. XLIST compares the current index into the line (STINDEX) to the displacement to the next statement (NXTSTD). If STINDEX is not pointing to the next statement, at least one line number is specified. In this case, XLIST calls a subroutine of Execute Expression to evaluate the line number and convert it to a positive integer, which XLIST stores in TSLNUM as the starting line number. If a second line number is specified, XLIST calls Execute Expression again and stores the value in LELNUM as the final line to LIST. If there is no second line number, then XLIST makes the ending line number equal to the starting line number, and only one line will be LISTed. If no line numbers were present, then TSLNUM and LELNUM still contain their default values, and all the program lines will be LISTed. XLIST gets the first line to be LISTed by calling the Execution Control subroutine GETSTMT to initialize the line parameters to correspond to the line number in TSLNUM. If we are not at the end of the Statement Table, and if the current line's number is less than or equal to the final line number to be LISTed, XLIST calls a subroutine :LLINE to list the line. After LISTing the line, XLIST calls Execution Control's subroutines to point to the next line. LISTing continues in this manner until the end of the Statement Table is reached or until the final line specified has been printed. When XLIST is finished, it exits via XRTN at $B719, which makes the LIST statement the current statement again and then returns to Execution Control.

LIST Subroutines
:LLINE ($BS5C) The LLINF routine LISTs the current line (the line whose address is in STMCUR). :LLINE gets the line number from the beginning of the tokenized line. The floating point package is called to convert the integer to floating point and then to printable ATASCII. The result is stored in the buffer indicated by INBUFF. :LLINE calls a subroutine to print the line number and then a blank. For every statement in the line, :LLINE sets STINDEX to point to the statement name token and calls the :LSTMT routine ($B590) to LIST the statement. When all statements have been LISTed, :LLINE returns to its caller, XLIST.

:LSTMT ($B590) The :LSTMT routine LISTs the statement which starts at the current displacement (in INDEX) into the current line. This routine does the actual language translation from tokens to BASIC statements. :LSTMT uses two subroutines, :LGCT and :LGNT, to get the current and next token, respectively. If the end of the statement has been reached, these routines both pull the return address of their caller off the 6502 CPU stack and return to :LSTMT's caller, :LLINE. Otherwise, they return the requested token from the tokenized statement line. The first token in a statement is the statement name token. :LSTMT calls a routine which prints the corresponding statement name by calling :LSCAN to find the entry and :LPRTOKEN to print it. In the discussion of the Program Editor we saw that an erroneous statement was given a statement name of ERROR and saved in the Statement Table. If the current statement is this ERROR statement or is REM or DATA, :LSTMT picks up each remaining character in the statement and calls PRCHAR ($BA9F) to print the character. Each type of token is handled differently. :LSTMT determines the type (variable, numeric constant, string constant, or operator) and goes to the proper code to translate it.

Variable Token. A variable token has a value greater than or equal to $80. When :LSTMT encounters a variable token, it turns off the most significant bit to get an index into the Variable Name Table. :LSTMT asks the :LSCAN routine to get the address of this entry. :LSTMT then calls :LPRTOKEN ($B535) to print the variable name. If the last character of the name is (, the next token is an array left parenthesis operator, and :LSTMT skips it.

Numeric Constant Token. A numeric constant is indicated by a token of $0E. The next six bytes are a floating point number. :LSTMT moves the numeric constant from the tokenized line to FRO ($D4) and asks the floating point package to convert it to ATASCII. The result is in a buffer pointed to by INBUFF. :LSTMT moves the address of the ATASCII number to SRCADR and tells :LPRTOKEN to print it.

String Constant Token. A string constant is indicated by a token of $0F. The next byte is the length of the string followed by the actual string data. Since the double quotes are not stored with a string constant, :LSTMT calls PRCHAR ($BA9F) to print the leading double quote. The string length tells :LSTMT how many following characters to print without translation. :LSTMT repeatedly gets a character and calls PRCHAR to print it until the whole string constant has been processed. It then asks PRCHAR to print the ending double quote.

Operator Token. An operator token is any token greater than or equal to $10 and less than $80. By subtracting $10 from the token value, :LSTMT creates an index into the Operator Name Table. :LSTMT calls :LSCAN to find the address of this entry. If the operator is a function (token value greater than or equal to $3D), :LPROTOKEN is called to print it. If this operator is not a function but its name is alphabetic (such as AND), the name is printed with a preceding and following blank. Otherwise, :LPRTOKEN is called to print just the operator name.

:LSCAN ($B50C) This routine scans a table until it finds the translation of a token into an ATASCII name. A token's value is based on its table entry number; therefore, the entry number can be derived by modifying the token. For example, a variable token is created by machine-language ORing the table entry number of the variable name with $80. The entry number can be produced by ANDing out the high-order bit of the token. It is this entry number, stored in SCANT, that the :LSCAN routine uses. The tables scanned by :LSCAN have a definite structure. Each entry consists of a fixed length portion followed by a variable length ATASCII portion. The last character in the ATASCII portion has the high-order bit on. Using these facts, :LSCAN in 5 he entry corresponding to the entry number in SCANT and puts the address of the ATASCII portion in SCRADR.

:LPRTOKEN ($B535) This routine's task is to print the string of ATASCII characters whose address is in SCRADR. :LPRTOKEN makes sure the most significant bit is off (except for a carriage return) and prints the characters one at a time until it has printed the last character in the string (the one with its most significant bit on).

Part XII

Atari Hardware Control Statements

Converted 2006 by Andreas Bertelmann for ABBUC

12 Atari Hardware Control Statements

The Atari Hardware Control Statements allow easy access to some of the computer's graphics and audio capabilities. The statements in this group are COLOR, GRAPHICS, PLOT, POSITION, DRAWTO, SETCOLOR, LOCATE, and SOUND.

XGR ($BA50) The GRAPHICS statement determines the current graphics mode. The XGR simulation routine executes the GRAPHICS statement. The XGR routine first closes IOCB 6. It then calls an Execute Expression subroutine to evaluate the graphics mode value and convert it to an integer. XGR sets up to open the screen by putting the address of a string "S." into INBUFF. It creates an AUX1 and AUX2 byte from the graphics mode integer. XGR calls a BASIC I/O routine which sets up IOCB 6 and calls CIO to open the screen for the specified graphics mode. Like all BASIC routines that do I/O, XGR jumps to the IOTEST routine, which determines what to do next based on the outcome of the I/O.

XCOLOR ($BA29) The COLOR statement is simulated by the XCOLOR routine. XCOLOR calls a subroutine of Execute Expression to evaluate the color value and convert it to an integer. XCOLOR saves this value (MOD 256) in BASIC memory location COLOR ($C8). This value is later retrieved by XPLOT and XDRAWTO.

XSETCOLOR ($89B7) The routine that simulates the SETCOLOR statement, XSETCOLOR, calls a subroutine of Execute Expression to evaluate the color register specified in the tokenized line. The Execute Expression routine produces a one-byte integer. If the value is not less than 5 (the number of color registers), XSETCOLOR exits via the Error Handling Routine at entry point ERVAL. Otherwise, it calls Execute Expression to get two more integers from the tokenized line. To calculate the color value, XSETCOLOR multiplies the tirst integer (MOD 256) byl6 and adds the second (MOD 256). Since the operating system's five color registers are in consecutive locations starting at $2C4 XSETCOLOR uses the register value specified as an index to the proper register location and stores the color value there.

XPOS ($BA16) The POSITION statement, which specifies the X and Y coordinates of the graphics cursor, is simulated by the XPOS routine. XPOS uses a subroutine of Execute Expression to evaluate the X coordinate of the graphics window cursor and convert it to an integer value. The two-byte result is stored in the operating system's X screen coordinate location (SCRX at $55). This is the column number or horizontal position of the cursor. XPOS then calls another Execute Expression subroutine to evaluate the Y coordinate and convert it to a one-byte integer. The result is stored in the Y screen coordinate location (SCRY at $54). This is the row number, or vertical position.

XLOCATE ($BC95) XLOCATE, which simulates the LOCATE statement, first calls XPOS to set up the X and Y screen coordinates. Next it initializes IOCB 6 and joins a subroutine of XGET to do the actual I/O required to get the screen data into the variable specified.

XPLOT ($5A76) XPLOT, which simulates the PLOT statement, first calls XPOS to set the X and Y coordinates of the graphics cursor. XPLOT gets the value that was saved in COLOR ($C8) and joins a PUT subroutine (PRCX at $BAA1) to do the I/O to IOCB 6 (the screen).

XDRAWTO ($BA31) The XDRAWTO routine draws a line from the current X,Y screen coordinates to the X,Y coordinates specified in the statement. The routine calls XPOS to set the new X,Y coordinates. It places the value from BASIC's memory location COLOR into OS location SVCOLOR ($2FB). XDRAWTO does some initialization of IOCB 6 specifying the draw command ($11). It then calls a BASIC I/O routine which finishes the

initialization of IOCB 6 and calls CIO to draw the line. Finally, XDRAWTO jumps to the IOTEST routine, which will determine what to do next based on the outcome of the I/O.

XSOUND ($K9DD) The Atari computer hardware uses a set of memory locations to control sound capabilities. The SOUND statement gives the user access to some of these capabilities. The XSOUND routine, which simulates the SOUND statement, places fixed values in some of the sound locations and user specified values in others. The XSOUND routine uses Execute Expression to get four integer values from the tokenized statement line. If the first integer (voice) is greater than or equal to 4, the Error Handling Routine is invoked at ERVAL. The OS audio control bits are all turned off by storing a 0 into $D208. Any bits left on from previous serial port usage are cleared by storing 3 in $D20F. The Atari has four sound registers (one for each voice) starting at $D200. The first byte of each two-byte register determines the pitch (frequency). In the second byte, the four most significant bits are the distortion, and the four least significant bits are the volume. The voice value mentioned earlier is multiplied by 2 and used as an index into the sound registers. The second value from the tokenized line is stored as the pitch in the first byte of one of the registers ($D200, $D202, $D204, or $D206), depending on the voice index. The third value from the tokenized line is multiplied by 16 and the fourth value is added to it to create the value to be stored as distortion/volume. The voice, times 2, is again used as an index to store this value in the second byte of a sound register ($D201, $D203, $D205, or $D207). The XSOUND routine then returns to Execution Control.

Part XIII

External Data I/O Statements

Converted 2006 by Andreas Bertelmann for ABBUC

13 External Data I/O Statements

The external data I/O statements allow data which is not part of the BASIC source program to flow into and out of BASIC. External data can come from the keyboard, a disk, or a cassette. BASIC can also create external information by sending data to external devices such as the screen, a printer, or a disk. The INPUT and GET statements are the primary statements used for obtaining information from external devices. The PRINT and PUT statements are the primary statements for sending data to external devices. XIO, LPRINT, OPEN, CLOSE, NOTE, POINT and STATUS are specialized I/O statements. LPRINT is used to print a single line to the "P:" device. The other statements assist in the I/O process.

XINPUT ($B316) The execution of the INPUT statement starts at)(INPUT ($B316).

Getting the Input Line. The first action of XINPUT is to read a line of data from the indicated device. A line is any combination of up to 255 characters terminated by the EOL character ($9B). This line will be read into the buffer located at $580. If the INPUT statement contained was followed by #<expression>, the data will be read from the IOCB whose number was specified by <expression>. If there was no #<expression>, IOCB 0 will be used. IOCB 0 is the screen editor and keyboard device (E:). If IOCB 0 is indicated, the prompt character (?) will be displayed before the input line request is made; otherwise, no prompt is displayed.

Line Processing. Once the line has been read into the buffer, processing of the data in that line starts at XINA ($B335). The input line data is processed according to the tokens in the INPUT (or READ) statements. These tokens are numeric or string variables separated by commas.

Processing a Numeric Variable. If the new token is a numeric variable, the CVAFP routine is called to convert the next characters in the input line to a floating point number. If this conversion does not report an error1 and if the next input line character is a comma or an EOL, the floatixig point value is processed. The processing of a valid numeric input value consists of calling RTNVAR to return the variable and its new value to the Variable Value Table. If there is an error, INPUT processing is aborted via the ERRINP routine. If there is no error, but the user has hit BREAK, the process is aborted via XSTOP. If there is no abort, XINX ($B389) is called to continue with INPUT's next task.

Processing a String Variable. If the next statement token is a string variable, it is processed at XISTR ($B35E). This routine is also used by the READ statement. If the calling statement is INPUT, then all input line characters from the current character up to but not including the EOL character are considered to be part of the input string data. If the routine was called by READ, all characters up to but not including the next comma or EOL are considered to be part of the input string. The process of assigning the data to the string variable is handled by calling RISASN ($B386). If RISASN does not abort the process because of an error like DIMENSION TOO SMALL, XINX is called to continue with INPUT's next task.

XINX. The XINX ($B389) routine is entered after each variable token in an INPUT or a READ statement is processed. If the next token in the statement is an EOL, the INPUT/READ statement processing terminates at XIRTS ($B3A1). XIRTS restores the line buffer pointer ($80) to the RAM table buffer. It then restores the enter device to IOCB 0 (in case it had been changed to some other input device). Finally, XIRTS executes an RTS instruction. If the next INPUT/READ statement token is a comma, more input data is needed. If the next input line character is an EOL, another input line is obtained. If the statement was INPUT, the new line

is obtained by entering XINO ($B326). If the statement was READ, the new lineis obtained by entering XRD3 ($B2D0). The processing of the next INPUT/READ statement variable token continues at XINA.

XGET ($BC7F) The GET statement obtains one character from some specified device and assigns that character to a scalar (non-array) numeric variable. The execution of GET starts at XGET ($BC7F) with a call to GIODVC. GIODVC will set the I/O device to whatever number is specified in the #< expression> or to IOCB zero if no #<expression> was specified. (If the device is IOCB 0 (E:), the user must type RETURN to force E: to terminate the input.) The single character is obtained by calling 103. The character is assigned to the numeric variable by calling ISVAR1 ($BD2D). ISVAR1 also terminates the GET statement processing.

PRINT The PRINT statement is used to transmit text data to an external device. The arguments in the PRINT statement are a list of numeric and/or string expressions separated by commas or semicolons. If the argument is numeric, the floating point value is converted to text form. If the argument is a string, the string value is transmitted as is, If an argument separator is a comma, the arguments are output in tabular fashion: each new argument starts at the next tab stop in the output line, with blanks separating the arguments. If the argument separator is a semicolon, the transmitted arguments are appended to each other without separation. The transmitted line is terminated with an EOL, unless a semicolon or comma directly precedes the statement's EOL or statement separator (:).

XPRINT ($B3B6). The PRINT routine begins at XPRINT ($B3B6). The tab value is maintained in the PTABW ($C9) cell. The cell is initialized with a value of ten during BASIC's cold start, so that commas in the PRINT line cause each argument to be displaced ten positions after the beginning of the last argument. The user may POKE PTABW to set a different tab value. XPRINT copies PTABW to SCANT ($AF). SCANT will be used to contain the next multiple-of-PTABW output line displacement - the column number of the next tab stop. COX is initialized to zero and is used to maintain the current output column or displacement. 97

XPRO. XPRINT examines the next statement token at XPRO ($B3BE), classifies it, and executes the proper routine.

Token. If the next token is #, XPRIOD ($B437) is entered. This routine modifles the list device to the device specified in the # <expression>. XPRO is then entered to process the next token.

, Token. The XPTAB ($B419) routine is called to process the , token. Its job is to tab to the next tab column. If COX (the current column) is greater than SCANT, we must skip to the next available tab position. This is done by continuously adding PTABW to SCANT until COX is less than or equal to SCANT. When COX is less than SCANT, blanks ($20) are transmitted to the output device until COX is equal to SCANT. The next token is then examined at XPRO.

EOL and: Tokens. The XPEOS ($B446) routine is entered for EOL and tokens. If the previous token was a; or, token, PRINT exits at XPRTN ($B458). If the previous token was not a; or, token, an EOL character is transmitted before exiting via XPRTN.

; Token. No special action is taken for the; token except to go to XPRO to examine the next token.

Numbers and Strings. If the next token is not one of the above tokens, Execute Expression is called to evaluate the expression. The resultant value is popped from the argument stack and its type is tested for a number or a string. If the argument popped was numeric, it will be

converted to text form by calling CVFASC. The resulting text is transmitted to the output device from the buffer pointed to by INBUFF ($F3). XPRO is then entered to process the next token. If the argument popped was a string, it will be transmitted to the output device by the code starting at :XPSTh ($B3F8). This code examines the argument parameters to determine the current length of the string. When the string has been transmitted, XPRO is entered to process the next token.

XLPRINT ($8464) LPRINT, a special form of the PRINT statement, is used to print a line to the printer device (P:). The XLPRINT routine starts at $B464 by opening IOCB 7 for output to the P: device. XPRINT is then called to do the printing. When the XPRINT is done, IOCB 7 is closed via CLSYS1 and LPRINT is terminated.

XPUT ($BC72) The PUT statement sends a single byte from the expression in the PUT statement to a specified external device. Processing starts at XPUT ($BC72) with a call to GIODVC. GIODVC sets the I/O device to the IOCB specified in #<expression>. If a #< expression> does not exist, the device will be set to IOCB zero (E:). The routine then calls GETINT to execute PUT's expression and convert the resulting value to a two-byte integer. The least significant byte of this integer is then sent to the PUT device via PRCX. PRCX also terminates the PUT processing.

XXIO ($BBE5) The XIO statement, a general purpose I/O statement, is intended to be used when no other BASIC I/O statement will serve the requirements. The XIO parameters are an IOCB I/O command, an IOCB specifying expression, an AUXi value, an AUX2 value, and finally a string expression to be used as a filespec parameter. XIO starts at XXIO ($BBE5) with a call to GIOCMD. CIOCMD gets the IOCB command parameter. XIO then continues at XOP1 in the OPEN statement code.

XOPEN ($BBEB) The OPEN statement is used to open an external device for input and/or output. OPEN has a # <expression>, the open type parameter (AUX1), an AUX2 parameter, and a string expression to be used as a filespec. OPEN starts at XOPFN at $BBEB. It loads the open command code into the A register and continues at XOP1.

XOP1. XOP1 continues the OPEN and XIO statement processing. It starts at $BBED by storing the A register into the IOCMD cell. Next it obtains the AUX1(open type) and AUX2 values from the statement. The next parameter is the filespec string. In order to insure that the filespec has a proper terminator, SETSEOL is called to place a temporary EOL at the end of the string. The XIO or OPEN command is then executed via a call to IO1. When IO1 returns, the temporary EOL at the end of the string is replaced with its previous value by calling RSTSEOL. OPEN and XIO terminate by calling TOTEST to insure that the command was executed without error.

XCLOSE ($BC1B) The CLOSE statement, which closes the specified device, starts at XCLOSE ($BCIB). It loads the IOCB close command code into the A register and continues at GDVCIO.

GDVCEO. GDVCIO ($BC1D) is used for general purpose device I/O. It stores the A register into the IOCMD cell, calls GIODVC to get the device from # <expression>, then calls IO7 to execute the IO0. When IO7 returns, IOTEST is called to test the results of the I/O and terminate the routine.

XSTATUS ($BC28) The STATUS statement executes the IOCB status command. Processing starts at XSTATUS ($BC28) by calling GIODVC to get the device number from # <expression>. It then calls IO8 with the status command in the A register. When IO8 returns, the status returned in the IOCB status cell is assigned to the variable specified in the STATUS statement by calling ISVARI. ISVAR1 also terminates the STATUS statement processing.

XNOTE ($BC3G) The NOTE statement is used specifically for disk random access. NOTE executes the Disk Device Dependent Note Command, $26, which returns two values representing the current position within the file for which the IOCB is open. NOTE begins at XNOTE at $BC36. The code loads the command value, $26, into the A register and calls GDVCIO to do the I/O operation. When GDVCIO returns, the values are moved from AUX3 and AUX4 to the first variable in the NOTE statement. The next variable is assigned the value from AUX5.

XPOINT ($8C4D) The POINT statement is used to position a disk file to a previously NOTEd location. Processing starts at XPOINT ($BC4D). This routine conyerts the first POINT parameter to an integer and stores the value in AUX3 and AUX4. The second parameter is then converted to an integer and its value stored in AUX5. The POINT command, $25, is executed by calling GDIO1, which is part of GDVCIO.

Miscellaneous I/O Subroutines IOTEST. IOTEST($BCB3) is a general purpose routine that examines the results of an I/O operarion. If the I/O processing has returned an error, IOTEST processes that error. IOTEST starts by calling LDIOSTA to get the status byte from the IOCB that performed the last I/O operation. If the byte value is positive (less than 128), IOTEST returns to the caller. If the status byte is negative, the I/O operation was abnormal and processing continues at SICKIO. If the I/O aborted due to a BREAK key depression, BRKBYT ($11) is set to zero to indicate BREAK. If a LOAD was in progress when BREAK was hit, exit is via COLDSTART; otherwise IOTEST returns to its caller. If the error was not from IOCB 7 (the device BASIC uses), the error status value is stored in ERRNUM and ERROR is called to print the error message and abort program execution. If the error was from IOCB 7, then IOCB 7 is closed and ERROR is called with the error status value in ERRNUM - unless ENTER was being executed, and the error was an end- of-file error. In this case, IOCB 7 is closed, the enter device is reset to IOCB 0, and SNX2 is called to return control to the Program Editor.

I/O Call Routine. All I/O is initiated from the routine starting at IO1 ($BD0A). This routine has eight entry points, IO1 through IO8, each of which stores predetermined values in an IOCB. All IOn entry points assume that the X register contains the IOCB value, times 16. IO1 sets the buffer length to 255. IO2 sets the buffer length to zero. IO3 sets the buffer length to the value in the Y register plus a most-significant length byte of zero. IO4 sets the buffer length from the values in the Y,A register pair, with the A register being the most-significant value. IO5 sets the buffer address from the value in the INBUFF cell (sF3). IO6 sets the buffer address from the Y,A register pair. The A register contains the most significant byte. IO7 sets the 1/0 command value from the value in the IOCMD cell. IO8 sets the I/O command from the value in the A register. All of this is followed by a call to the operating system CIO entry point. This call executes the I/O. When CIO returns, the general I/O routine returns to its caller. 102

Part XIV

Internal I/O Statements

14 Internal I/O Statements

The READ, DATA, and RESTORE statements work together to allow the BASIC user to pass predetermined information to his or her program. This is, in a sense, internal I/O.

XDATA ($A9E7) The information to be passed to the BASIC program is stored in one or more DATA statements. A DATA statement can occur any place in the program, but execution of a DATA statement has no effect. When Execution Control encounters a DATA statement, it expects to process this statement just like any other. Therefore an XDATA routine is called, but XDATA simply returns to Execution Control.

XREAD ($B283) The XREAD routine must search the Statement Table to find DATA. It uses Execution Control's subroutines and line parameters to do this. When XREAD is done, it must restore the line parameters to point to the READ statement. In order to mark its place in the Statement Table, XREAD calls a subroutine of XGOSUB to put a GOSUB-type entry on the Runtime Stack. The BASIC program may need to READ some DATA, do some other processing, and then READ more DATA. Therefore, XREAD needs to keep track of just where it is in which DATA statement. There are two parameters that provide for this. DATALN ($B7) contains the line number at which to start the search for the next DATA statement. DATAD ($B6) contains the displacement of the next DATA element in the DATALN line. Both values are set to zero as part of RUN and CLR statement processing. XREAD calls Execution Control's subroutine GETSTMT to get the line whose number is stored in DATALN. If this is the first READ in the program and a RESTORE has not set a different line number, DATALN contains zero, and GETSTMT will get the first line in the program. On subsequent READs, GETSTMT gets the last DATA statement that was processed by the previous READ. After getting its first line, XREAD calls the XRTN routine to restore Execufion Control's line parameters. The current line number is stored in DATALN. XREAD steps through the line, statement by statement, looking for a DATA statement. If the line contains no DATA statement, then subsequent lines and statements are examined until a DATA statement is found. When a DATA statement has been found, XREAD inspects the elements of the DATA statement until it finds the element whose displacement is in DATAD. If no DATA is found, XREAD exits via the ERROOD entry point in the Error Handling Routine. Otherwise, a flag is set to indicate that a READ is being done, and XREAD joins XINPUT at :XINA. XINPUT handles the assignment of the DATA values to the variables. (See Chapter 13.)

XREST ($B268) The RESTORE statement allows the BASIC user to re-READ a DATA statement or change the order in which the DATA statements are processed. The XREST routine simulates RESTORE. XREST sets DATALN to the line number given, or to zero if no line number is specified. It sets DATAD to zero, so that the next READ after a RESTORE will start at the first element in the DATA line specified in DATALN.

Part XV

Miscellaneous Statements

Converted 2006 by Andreas Bertelmann for ABBUC

15 Miscellaneous Statements

XDEG ($B261) and XRAD ($B266) The transcendental functions such as SIN or COS will work with either degrees or radians depending on the setting of RADFLG ($FB). The DEG and'RAD statements cause RADFLG to be set. These statements are simulated by the XDEG and XRAD routines, respectiveW. The XDEG routine stores a six in RADI'LG. XRAD sets it to zero. These particular values were chosen because they aid the transcendental functions in their calculations. RADFLG is set to zero during BASIC's initialization process and also during simulation of the RUN statement.

XPOKE ($B24C) The POKE statement is simulated by the XPOKE routine. XPOKE calls a subroutine of Execute Expression to get the address and data integers from the tokenized line. XPOKE then stores the data at the specified address.

XBYE ($A9E8) The XBYE routine simulates the BYE statement. XBYE closes all IOCBs (devices and files) and then jumps to location $E471 in the Operating System. This ends BASIC and causes the memo pad to be displayed.

XDOS ($A9EE) The DOS statement is simulated by the XDOS routine. The XDOS routine closes all IOCBs and jumps to whatever address is stored in location $0A. This will be the address of DOS if DOS has been loaded. If DOS has not been loaded, $0A will point to the memo pad.

XLET ($AAE0) The LET and implied LET statements assign values to variables. They both invoke the XLET routine, which consists of the Execute Expression routines. (See Chapter 7.)

XREM ($A9E7) The REM statement is for documentation purposes only and has no effect on the running program. The routine which simulates REM, XREM, simply executes an RTS instruction to return to Execution Control.

XERR ($B91E) When a line containing a syntax error is entered, it is given a special statement name token to indicate the error. The entire line is flagged as erroneous no matter how many previously good statements are in the line. The line is then stored in the Statement Table. The error statement is processed just like any other. Execution Control calls a routine, XERR, which is one of the entry points to the Error Handling Routine. It causes error 17 (EXECUTION OF GARBAGE).

XDIM ($B1D9) The DIMension statement, simulated by the XDIM routine, reserves space in the String/Array Table for the DIMensioned variable. The XDIM routine calls Execute Expression to get the variable to be DIMensioned from the Variable Value Table. The variable entry is put into a work area. In the process, Execute Expression gets the first and second DIMension values and sets a default of zero if only one value is specified. XDIM checks to see if the variable has already been DIMensioned. If the variable was already DIMensioned, XDIM exits via the ERRDIM entry point in the Error Handling Routine. If not, a bit is set in the variable type byte in the work area entry to mark this variable as DIMensioned. Next, XDIM calculates the amount of space required. This calculation is handled differently for strings and arrays.

DIMensioning an Array. XDIM first increments both dimension values by one and then multiplies them together to get the number of elements in the array. XDIM multiplies the result by 6 (the length of a floating point number) to get the number of bytes required. EXPAND is called to expand the String/Array Table by that amount. XDIM must finish building the variable entry in the work area. It stores the first and second dimension values in the entry. It also stores the array's displacement

into the String/Array Table. It then calls an Execute Expression subroutine to return the variable to the Variable Value Table. (See Chapter 3.)

DIMensioning a String. Reserving space for a string in the String/Array Table is much simpler. XDIM merely calls the EXPAND routine to expand by the user-specified size. XDIM must also build the Variable Value Table entry in the work area. It sets the current length to 0 and the maximum length to the DIMensioned value. The displacement of the string into the String/Array Table is also stored in the variable. XDIM then calls a subroutine of Execute Expression to return the variable entry to the Variable Value Table. (See Chapter 3.)

Part XVI

Initialization

16 Initialization

When the Atari computer is powered up with the BASIC cartridge in place, the operating system does some processing and then jumps to a BASIC routine. Between the time that BASIC first gets control and the time it prints the READY message, initialization takes place. This initialization is called a cold start. No data or tables are preserved during a cold start. Initialization is repeated if things go terribly awry. For example, if there is an I/O error while executing a LOAD statement, BASIC is totally confused. It gives up and begins all over again with the COLDSTART routine. Sometimes a less drastic partial initialization is necessary. This process is handled by the WARMSTART routine, in which some tables are preserved. Entering the NEW statement, simulated by the XNEW routine, has almost the same effect as a cold start.

COLDSTART ($A000) Two flags, LOADFLG and WARMFLG, are used to determine if a cold or warm start is required. The load flag, LOADFLG ($CA), is zero except during the execution of a LOAD statement. The XLOAD routine sets the flag to non-zero when it starts processing and resets it to zero when it finishes. If an I/O error occurs during that interval, IOTEST notes that LOADFLG is non-zero and jumps to COLDSTART. The warm-start flag, WARMFLG ($08), is never set by BASIC. It is set by some other routine, such as the operating system or DOS. If WARMFLG is zero, a cold start is done. If it is non-zero, a warm start is done. During its power-up processing, before BASIC is given control, OS sets WARMFLG to zero to request a cold start. During System Reset processing, OS sets the flag to non-zero, indicating a warm start is desired. If DOS has loaded any data into BASIC's program area during its processing, it will request a cold start. The COLDSTART routine checks both WARMFLG and LOADFLG to determine whether to do a cold or warm start. If a cold start is required, COLDSTART initializes the 6502 CPU stack and clears the decimal flag. The rest of its processing is exactly the same as if the NEW statement had been entered.

XNEW ($A00C) The NEW statement is simulated by the XNEW routine. XNEW resets the load flag, LOADFLG, to zero. It initializes the zero- page pointers to BASIC's RAM tables. It reserves 256 bytes at the loW memory address for the multipurpose buffer and stores its address in the zero-page pointer located at $80. Since none of the RAM tables are to retain any data, their zero- page pointers ($82 through $90) are all set to lOW memory plus 256. The Variable Name Table is expanded by one byte, which is set to zero. This creates a dummy end-of-table entry. The Statement Table is expanded by three bytes. The line number of the direct statement ($8000) is stored there along with the length (three). This marks the end of the Statement Table. A default tab value of 10 is set for the PRINT statement.

WARMSTART ($A04D) A warm start is the least drastic of the three types of initialization. Everything the WARMSTART routine does is also done by COLDSTART and XNEW. The stop line number (STOPLN), the error number (ERRNUM), and the DATA parameters (DATALN and DATAD) are all set to zero. The RADFLG flag is set to zero, indicating that transcendental functions are working in radians. The break byte (BRKBYT) is set off and $FF is stored in TRAPLN to indicate that errors are not being trapped. All IOCBs (devices and files) are closed. The enter and list devices (ENTDTD and LISTDTD) are set to zero to indicate the keyboard and the screen, respectively. Finally, the READY message is printed and control passes to the Program Editor.

Index

- 6 -

6502 3

- A -

Absolute Non-Terminal Vector 26
Application high memory 13
Argument Stack 40
Array 40
Array Address 40
Array Entry 13
Assembler 3
Atari BASIC 3
Atari keyboard 21

- B -

BASIC 49, 76
BYE 73

- C -

Cartridge 76
Change Last Token 26
CIX 21
CLOAD 56
CLOSE 66
COLDSTART 76
COLOR 63
Constants 26
CONT 49
COS 73
COX 21
CSAVE 56

- D -

DATA 71
Deleting lines 21
DIM 40, 73
DIMensioning a String 73
DIMensioning an Array 73
DIRFLG 21
DOS 73
DRAWTO 63

- E -

ECHNG 26
End-of-line 36
ENTER 59
ERNTV 26
Error Handling Routine 49
Execute Expression 8
Execution Control 36, 52
Expression Non-Terminal Vector 26
Expression Rearrangement 40
External Subroutine Call 26

- F -

Faster Programming or Faster Programs? 3
Flags 21
FOR 52

- G -

GDVCEO 66
GET 66
GETADR 26
GETLL 36
GETSTMT 36
GETTOK 52
GNXTL 36
GOSUB 52
GRAPHICS 63

- H -

Hardware Control Statements 63
High Level Languages 3
High memory address 13

- I -

I/O Call Routine 66
IF 52
Indices 21
Initialization 76
Initialization Execution 36
Initializing 21
INPUT 66

- L -

Language statements 49
Language Translation Methods 3
Language Translators 3
LET 73
LIFO Stacks 8
Line Processing 21
LIST 59
LOAD 56
LOADFLG 76
LOCATE 63
Low memory address 13
LPRINT 66
LPRTOKEN 59
LSCAN 59
LSTMT 59

- M -

MAXCIX 21
Memory Management Routines 13
Memory usage 13
Multipurpose Buffer 13

- N -

NEW 76
NEXT 26, 52
NOTE 66
Numeric Constant 59
numerical grouping 8

- O -

ON 52
OPEN 66
Operator 59
Operator Executions 40
Operator Precedence Table 40
Operator Stack 40
Operator Token 26
Operators 26

- P -

PLOT 63
POINT 66
POKE 73
POP 26
POPRSTK 52
POSITION 63
powered up 76
Pre-compiler 8, 26
Pre-compiling Interpreter 3
PRINT 66
Problems with the Compiler Method 3
Program Constructor 8
Program Editor 8
Program Executor 8
Program Termination 49
PSHRSTK 52
PUSH 26
PUT 66

- R -

RAM Tables 8
READ 66, 71
Relative Non-Terminal Vectors 26
REM 73
RESTORE 71
RETURN 52
ROM Tables 8
Runtime Stack 13

- S -

SAVE 56
SAVE File Format 56
SEARCH 26
SETCODE 26
SETCOLOR 63
SETLINE 36
SETLN1 36
Simple Flow Control 52
SIN 73
SOUND 63
SRCONT 26
Stack 40
Statement Execution 36
Statement Table 13
STATUS 66
STMCUR 36
STOP 49
String 40
String Constant 59

String Entry 13
String/Array Table 13
SVVNTP 21
SVVYTE 21

- T -

Tables 8
TERMTST 26
TNCON 26
TNVAR 26
Token 59
Tokens 8
TRAP 52
TSCON 26
TSLNUM 36
TSVAR 26

- V -

Variable 59
Variable Name Table 13
Variable Value Table 13
Variables 26

- W -

WARMFLG 76
WARMSTART 76

- X -

XBYE 73
XCLOSE 66
XCOLOR 63
XCONT 49
XDATA 71
XDEG 73
XDIM 73
XDOS 73
XDRAWTO 63
XEND 49
XENTER 59
XERR 73
XFOR 52
XGET 66
XGOSUB 52
XGR 63
XIF 52
XINPUT 66
XINX 66
XIO 66
XLET 73
XLIST 59
XLOAD 56
XLOCATE 63
XLPRINT 66
XNEW 76
XNEXT 52
XNOTE 66
XON 52
XOP1 66
XOPEN 66
XPLOT 63
XPOINT 66
XPOKE 73
XPOP 52
XPOS 63
XPRINT 66
XPRO 66
XPUT 66
XRAD 73
XREAD 71
XREM 73
XREST 71
XRTN 52
XRUN 49
XSAVE 56
XSETCOLOR 63
XSOUND 63
XSTATUS 66
XSTOP 49
XTRAP 52
XXIO 66

- Z -

Zero Page Table Pointers 13